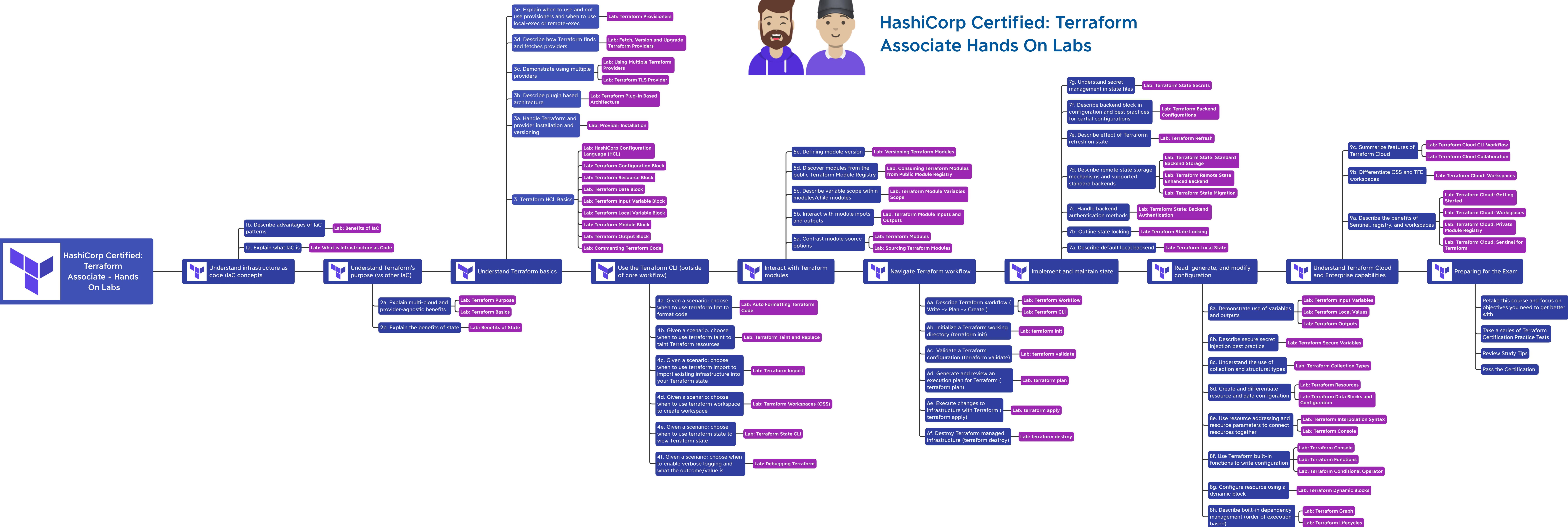


# HashiCorp Certified: Terraform Associate Hands On Labs





## Lab: What is Infrastructure as Code?

### What is Infrastructure as Code

Infrastructure as Code is essentially a hub that can be used for collaboration across the IT organization to improve infrastructure deployments, increase our ability to scale quickly, and improve the application development process. Infrastructure as Code allows us to do all this consistently and proficiently. By using Infrastructure as Code for both our on-premises infrastructure and the public cloud, our organization can provide dynamic infrastructure to both our internal team members and ensure our customers have an excellent experience.

### Benefits of IaC

While there are many benefits of Infrastructure as Code, a few key benefits include simplifying cloud adoption, allowing us to adopt cloud-based services and offerings to improve our capabilities quickly. Infrastructure as Code allows us to remove many of the manual steps required today for infrastructure requests, giving us the ability to automate approved requests without worrying about tickets sitting in a queue. We can also use Infrastructure as Code to provide capacity-on-demand by offering a library of services for our developers. We can publish a self-service capability where developers and application owners can be empowered to request and provision infrastructure that better matches their requirements. Again, all of this is possible while driving standardization and consistency throughout the organization, which can drive efficiencies and reduce errors or deviations from established norms.

### Example of IaC

#### IaC Tools

The list below represents some of the most popular Infrastructure as Code tools used by many organizations worldwide. These tools focus on deploying infrastructure on a private or public cloud platform. The list does NOT include tools such as Puppet, Chef, Saltstack, or Ansible since those are commonly placed in the configuration management category and don't really deploy infrastructure resources. There are likely other tools available, but they are not as popular as the ones listed below.

- HashiCorp Terraform - [terraform.io](https://www.terraform.io)
- AWS CloudFormation - [aws.amazon.com/cloudformation](https://aws.amazon.com/cloudformation)
- Azure Resource Manager (ARM) - [azure.microsoft.com](https://azure.microsoft.com)





- Google Cloud Deployment Manager - [cloud.google.com/deployment-manager/docs](https://cloud.google.com/deployment-manager/docs)
- Pulumi - [pulumi.com](https://pulumi.com)





## Lab: Benefits of IaC

### Benefits of IaC

While there are many benefits of Infrastructure as Code, a few key benefits include the simplification of cloud adoption, allowing us to quickly adopt cloud-based services and offerings to improve our capabilities. Infrastructure as Code allows us to remove many of the manual steps required today for infrastructure requests, giving us the ability to automate approved requests without worrying about tickets sitting in a queue. We can also use Infrastructure as Code to provide capacity on-demand by offering a library of services for our developers, even publishing a self-service capability where developers and application owners can be empowered to request and provision infrastructure that better matches their requirements. Again, all of this is possible while driving standardization and consistency throughout the organization, which can drive efficiencies and reduce errors or deviations from established norms.

### References

Infrastructure as Code in a Private or Public Cloud

## Lab Instructions

You have been tasked with deploying some basic infrastructure on AWS to host a proof of concept environment. The architecture needs to include both public and private subnets and span multiple Availability Zones to test failover and disaster recovery scenarios. You expect to host Internet-facing applications. Additionally, you have other applications that need to access the Internet to retrieve security and operating system updates.

- **Task 1:** Create a new VPC in your account in the US-East-1 region
- **Task 2:** Create public and private subnets in three different Availability Zones
- **Task 3:** Deploy an Internet Gateway and attach it to the VPC
- **Task 4:** Provision a NAT Gateway (a single instance will do) for outbound connectivity
- **Task 5:** Ensure that route tables are configured to properly route traffic based on the requirements
- **Task 6:** Delete the VPC resources
- **Task 7:** Prepare files and credentials for using Terraform to deploy cloud resources
- **Task 8:** Set credentials for Terraform deployment
- **Task 9:** Deploy the AWS infrastructure using Terraform

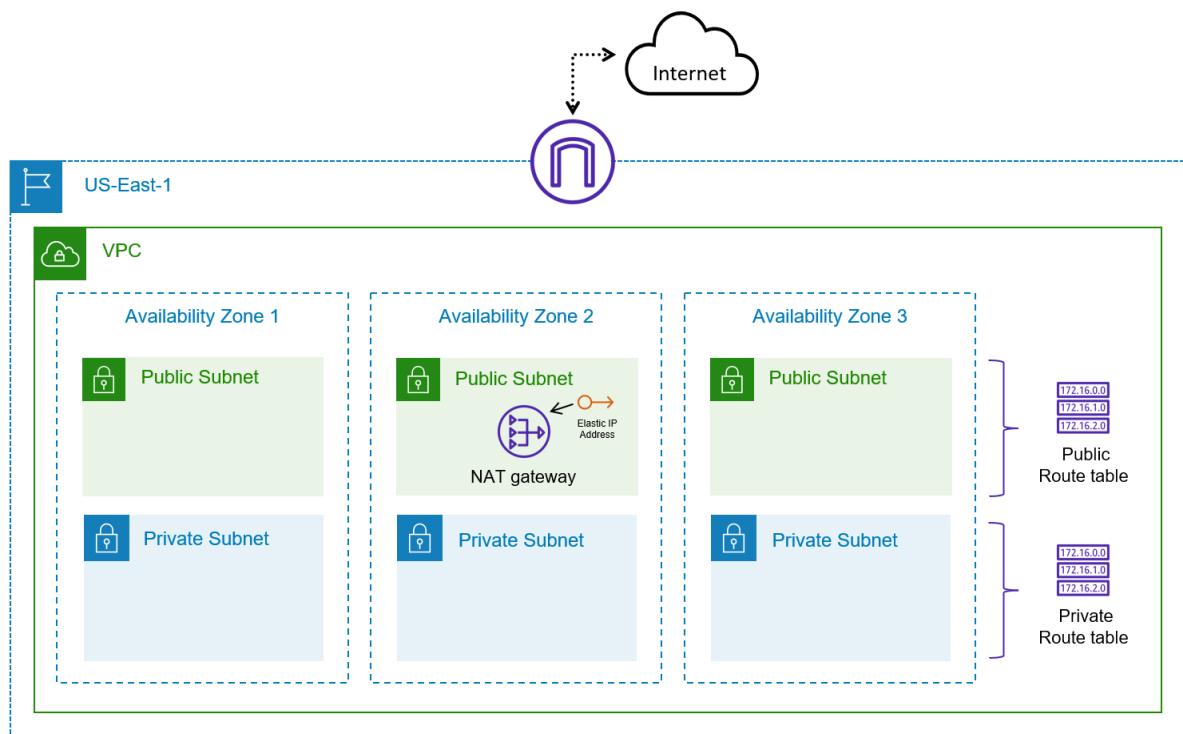




## Hands-On Labs

- **Task 10:** Delete the AWS resources using Terraform to clean up our AWS environment

The end state of the AWS environment should look similar to the following diagram:



**Figure 1:** Desired Infrastructure

This lab will walk you through configuring the infrastructure step by step using a manual process. After manually completing the tasks, the lab will show you how Terraform can be used to automate the creation of the same infrastructure.

### Task 1: Log into the AWS Console and create a VPC.

#### Step 1.1

In the VPC console, click **Create VPC**:





## Hands-On Labs

The screenshot shows the AWS VPC Management Console. On the left, there's a sidebar with options like 'VPC Dashboard', 'EC2 Global View', and 'Your VPCs'. The main area is titled 'Your VPCs' and shows a table with columns for Name, VPC ID, State, and IPv4 CIDR. There's a search bar at the top of the table and a 'Create VPC' button in the top right corner. A red arrow points to the 'Create VPC' button.

**Figure 2:** Create VPC

### Step 1.2

Give the VPC a name of **demo-vpc** and set the IPv4 CIDR block to use **10.0.0.0/16**. Leave all of the other settings as default and select **Create VPC** at the bottom of the Create VPC screen.





## Hands-On Labs

The screenshot shows the 'Create VPC' wizard in the AWS VPC Management Console. The 'VPC settings' section is visible, containing fields for a name tag and an IPv4 CIDR block. Red arrows point to the 'Name tag' field, which contains 'demo-vpc', and the 'IPv4 CIDR block' field, which contains '10.0.0.0/16'. Other sections like 'IPv6 CIDR block' and 'Tenancy' are also shown.

**Figure 3:** Configure VPC

### Task 2: Create public and private subnets in three different Availability Zones.

#### Step 2.1

In the VPC console, select **Subnets** from the left navigation panel. Click **Create Subnet**.





## Hands-On Labs

The screenshot shows the AWS VPC Management Console with the URL [console.aws.amazon.com/vpc/home?region=us-east-1#subnets](https://console.aws.amazon.com/vpc/home?region=us-east-1#subnets). The left sidebar is expanded, showing the 'VIRTUAL PRIVATE CLOUD' section with options like 'Your VPCs' (highlighted with a red arrow labeled '1'), 'Route Tables', 'Internet Gateways', 'Egress Only Internet Gateways', 'Carrier Gateways', 'DHCP Options Sets', and 'Elastic IPs'. The main content area is titled 'Subnets Info' and shows a table with columns 'Name', 'Subnet ID', and 'State'. A red arrow labeled '2' points to the 'Create subnet' button in the top right corner of the table header.

**Figure 4:** Select Subnet

### Step 2.2

Select the VPC created in Step 1 from the dropdown list. Give the subnet the name `private-subnet-1` and select `us-east-1a` from the dropdown list for the Availability Zone. Enter the IPv4 CIDR block of `10.0.0.0/24`.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



A screenshot of the AWS VPC Management Console. The top navigation bar shows 'VPC Management Console' and the URL 'console.aws.amazon.com/vpc/home?region=us-east-1#CreateSubnet'. The main area is titled 'VPC' and shows a dropdown menu for 'VPC ID' containing 'vpc-001fc9d9e20ebdf14 (demo-vpc)'. Below this is a section for 'Associated VPC CIDRs' with 'IPv4 CIDRs' set to '10.0.0.0/16'. The central part of the screen is titled 'Subnet settings' and shows 'Subnet 1 of 1'. It includes fields for 'Subnet name' (containing 'private-subnet-1'), 'Availability Zone' (set to 'US East (N. Virginia) / us-east-1a'), and 'IPv4 CIDR block' (set to 'Q 10.0.0.0/24'). A red arrow points to each of these three fields. At the bottom of the subnet configuration, there is a 'Tags - optional' section with a single tag 'Name: private-subnet-1'. The footer of the page includes links for 'Feedback', 'English (US)', 'Privacy Policy', 'Terms of Use', and 'Cookie preferences'.

**Figure 5:** Create Subnets

### Step 2.3

Repeat the previous step to create the additional subnets required to build out the required infrastructure, including 2 additional private subnets and the 3 public subnets. Use the following information to complete this step:

Subnet Name	Availability Zone	CIDR Block
private-subnet-2	us-east-1b	10.0.1.0/24
private-subnet-3	us-east-1c	10.0.2.0/24
public-subnet-1	us-east-1a	10.0.100.0/24
public-subnet-2	us-east-1b	10.0.101.0/24





Subnet Name	Availability Zone	CIDR Block
public-subnet-3	us-east-1c	10.0.102.0/24

### Task 3: Deploy an Internet Gateway and attach it to the VPC.

#### Step 3.1

In the VPC console, select Internet Gateways from the left navigation panel in the VPC console. Click the Create Gateway button in the top right of the AWS console.

The screenshot shows the AWS VPC Internet Gateways page. On the left, there's a sidebar with various VPC-related options like 'VPC Dashboard', 'EC2 Global View', and 'Route Tables'. Under 'Internet Gateways', a red arrow labeled '1' points to the link. At the top right, there's a prominent orange button labeled 'Create internet gateway'. A red arrow labeled '2' points to this button. The main area of the page displays a table with columns for 'Name', 'Internet gateway ID', 'State', and 'VPC ID', with a message stating 'No internet gateways found in this Region'.

Figure 6: IGW

#### Step 3.2

Give the new Internet Gateway a name of **demo-igw** and click the Create internet gateway button.





*Note: The Internet Gateway can incur charges on your account. For the purposes of this lab, you will incur very minimal charges, likely a penny or two (US dollars). Don't fret... we will delete this resource shortly.*

A screenshot of the AWS VPC console showing the 'Create internet gateway' wizard. The 'Internet gateway settings' step is displayed. A red arrow points to the 'Name tag' input field, which contains the value 'demo-igw'. Below this, the 'Tags - optional' section shows a single tag named 'Name' with the value 'demo-igw'. At the bottom right are 'Cancel' and 'Create internet gateway' buttons.

Create internet gateway Info

An internet gateway is a virtual router that connects a VPC to the internet. To create a new internet gateway specify the name for the gateway below.

**Internet gateway settings**

Name tag  
Creates a tag with a key of 'Name' and a value that you specify.

demo-igw (arrow pointing here)

**Tags - optional**  
A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

Key Value - optional

Name demo-igw Remove

Add new tag

You can add 49 more tags.

Cancel Create internet gateway

**Figure 7:** Create IGW

### Step 3.3

In the Internet Gateway console, select the Actions menu and choose **Attach to VPC** from the dropdown list.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

A screenshot of the AWS VPC Management Console. The URL is console.aws.amazon.com/vpc/home?region=us-east-1#InternetGateway/internetGatewayId=igw-072539ca96c72d21f. The page shows a success message: "The following internet gateway was created: igw-072539ca96c72d21f. You can now attach to a VPC to enable the VPC to communicate with the internet." Below this is the Internet Gateway details for "igw-072539ca96c72d21f / demo-igw". The "Actions" menu is open, with a red arrow pointing to the "Attach to VPC" option. The "Details" section shows the Internet gateway ID (igw-072539ca96c72d21f), State (Detached), VPC ID (-), and Owner (603991114860). The "Tags" section shows a single tag named "Name" with the value "demo-igw".

The following internet gateway was created: igw-072539ca96c72d21f. You can now attach to a VPC to enable the VPC to communicate with the internet.

igw-072539ca96c72d21f / demo-igw

Internet gateway ID	State	VPC ID	Owner
igw-072539ca96c72d21f	Detached	-	603991114860

Tags

Key	Value
Name	demo-igw

**Figure 8:** Attach IGW

Select the VPC created in Step 1 by clicking the text box and choosing the VPC. Click the **Attach internet gateway** button to complete the task.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs

A screenshot of the AWS VPC Attach Internet Gateway dialog box. The URL in the browser bar is `console.aws.amazon.com/vpc/home?region=us-east-1#AttachInternetGateway:internetGatewayId=igw-072539ca96c72d21f`. The main title is "Attach to VPC (igw-072539ca96c72d21f)". A red arrow points to a search input field containing the text "vpc-001fc9d9e20ebdf14". At the bottom right are "Cancel" and "Attach internet gateway" buttons.

**Figure 9:** Attach IGW to VPC

### Task 4: Provision a NAT Gateway (a single instance will do) for outbound connectivity.

#### Step 4.1

Back in the VPC Console, select **NAT Gateways** from the left navigation panel. Click the **Create NAT Gateway** button on the top right of the AWS console.





## Hands-On Labs

The screenshot shows the AWS VPC Management console. On the left, there's a sidebar with various VPC-related options. Under 'VIRTUAL PRIVATE CLOUD', 'NAT Gateways' is highlighted with a red arrow labeled '1'. At the top right of the main content area, there's a prominent orange button labeled 'Create NAT gateway' with a red arrow labeled '2' pointing to it. The main area displays a table with columns for Name, NAT gateway ID, Connectivity, State, and State message.

**Figure 10:** NAT Gateway Creation

### Step 4.2

Provide the name **demo-nat-gateway** for the NAT Gateway. Select a subnet by choosing **public-subnet-2** from the dropdown list. Keep the Connectivity Type as **Public**. Click the **Allocate Elastic IP** button to automatically create and assign a new Elastic IP address for the NAT Gateway. Scroll down and click the **Create NAT Gateway** button to complete the task.





The screenshot shows the 'Create NAT gateway' wizard in the AWS VPC Management Console. The 'Name' field is set to 'demo-nat-gateway'. The 'Subnet' dropdown is set to 'subnet-071d82e4018a288a3 (public-subnet-2)'. The 'Connectivity type' section has 'Public' selected. The 'Elastic IP allocation ID' dropdown is set to 'eipalloc-0e71989cda0c9350'. A red arrow points to the 'Allocate Elastic IP' button.

**Figure 11:** Create NAT Gateway

## Task 5: Ensure that route tables are configured to properly route traffic based on the requirements.

### Step 5.1

In the VPC console, select **Route Tables** from the left navigation panel. Click the **Create route table** button on the top right of the AWS console.

Provide the name **public-rtb** for the route table and select the VPC created in Step 1. Click the **Create route table** button to create your first route table.





## Hands-On Labs

The screenshot shows the 'Create route table' wizard in the AWS VPC Management Console. The 'Route table settings' section is visible, featuring a 'Name - optional' field containing 'public-rtb' and a 'VPC' dropdown menu showing 'vpc-001fc9d9e20ebdf14 (demo-vpc)'. Below this, the 'Tags' section allows adding a tag for 'Name' with value 'public-rtb'. At the bottom right, there are 'Cancel' and 'Create route table' buttons.

**Figure 12:** Create rtb

Repeat the above task to create a second route table. Name the second route table **private-rtb**. Select the same VPC created in Step 1. Click the **Create route table** button to create the second route table.

### Step 5.2

From the Route Tables console, select the tick box next to the route table named **public-rtb**. In the bottom panel, select the **Subnet Associations** tab. Click the **Edit subnet associations** button.





## Hands-On Labs

The screenshot shows the AWS VPC Management Route Tables page. A success message at the top states: "Route table rtb-02e9e1b3351eb50a3 | public-rtb was created successfully." The main table lists two route tables: "rtb-0527adaac20fccb2e" and "rtb-02e9e1b3351eb50a3". The second row, "rtb-02e9e1b3351eb50a3 / public-rtb", has its checkbox checked. Red arrow 1 points to the "Name" column. Red arrow 2 points to the "Subnet associations" tab, which is currently selected. Red arrow 3 points to the "Edit subnet associations" button.

**Figure 13:** Create rtb

Select the three **public** subnets from the list of available subnets by checking the tick box next to the subnets. These are the same three subnets that you created in Step 2. Once you have selected the three subnets, click on the **Save associations** button to save your configuration.

Repeat this step for the **private-rtb** and selecting the 3 private subnets that were created in Step 2.





## Hands-On Labs

The screenshot shows the AWS VPC Management Console with the URL <https://console.aws.amazon.com/vpc/home?region=us-east-1#EditRouteTableSubnetAssociations:RouteTableId=rtb-02e9e1b3351eb50a3>. The page title is "Edit subnet associations".

**Available subnets (3/3)**

Name	Subnet ID	IPv4 CIDR	IPv6 CIDR	Route table ID
public-subnet-3	subnet-0a43f67f788229c0e	10.0.102.0/24	-	Main (rtb-0527adaac20fccb2e)
public-subnet-2	subnet-071d82e4018a288a3	10.0.101.0/24	-	Main (rtb-0527adaac20fccb2e)
public-subnet-1	subnet-0229539d18b2daedf	10.0.100.0/24	-	Main (rtb-0527adaac20fccb2e)

**Selected subnets**

- subnet-0229539d18b2daedf / public-subnet-1
- subnet-071d82e4018a288a3 / public-subnet-2
- subnet-0a43f67f788229c0e / public-subnet-3

Buttons at the bottom: Cancel, Save associations

**Figure 14:** Create rtb

### Step 5.3

Now that the subnets have been associated with the proper route table, we need to add the routes to ensure network traffic is routed correctly. From the Route Tables console, select the **public-rtb** again. In the bottom pane, select the **Routes** tab and click **Edit Routes**.





## Hands-On Labs

The screenshot shows the AWS VPC Management Route Tables page. On the left, there's a sidebar with various VPC-related options like VPC Dashboard, EC2 Global View, and Route Tables. The Route Tables section is expanded, showing 'Route tables (1/2)'. In the main area, a table lists two route tables: 'rtb-0527adaac20fcbb2e' and 'rtb-02e9e1b3351eb50a3'. The second row, 'rtb-02e9e1b3351eb50a3 / public-rtb', has a checked checkbox in its first column. Red arrows labeled '1', '2', and '3' point to this row, the 'Routes' tab, and the 'Edit routes' button respectively. The 'Routes' tab is selected, showing one route entry: 'Destination: 10.0.0.0/16, Target: local, Status: Active, Propagated: No'.

**Figure 15:** Create rtb

In the Edit Routes window, click the **Add route** button. Enter **0.0.0.0/0** in the **Destination** text box to define our new route destination. Click the text box for **Target**, select **Internet Gateway**, and select the Internet Gateway that was created in Step 3. It should be the only one listed. Click **Save changes** to save the new route configuration.





The screenshot shows the AWS VPC Management Console with the URL <https://console.aws.amazon.com/vpc/home?region=us-east-1#EditRoutes:RouteTableId=rtb-02e9e1b3351eb50a3>. The page title is "Edit routes". There are two sections for editing routes:

- Top Section:** Destination 10.0.0.0/16, Target local, Status Active.
- Bottom Section:** Destination 0.0.0.0/0, Target igw-072539ca96c72d21f, Status -.

Buttons include "Remove", "Add route", "Cancel", "Preview", and a prominent orange "Save changes" button.

**Figure 16:** Create rtb

Repeat this step to add a route to the **private-rtb**. The Destination should be **0.0.0.0/0**. Click the text box for **Target**, select **NAT Gateway**, and choose the NAT Gateway that was created in Step 4. It should be the only one listed. Click **Save changes** to save the new route configuration.

---

**Congratulations, you have manually configured all of the resources and have an environment that now matches the desired configuration as stated by the requirements.**

---

Wasn't that fun? While this example given in the lab isn't too bad, how would you feel repeating this process across 10s or 100s of AWS accounts in your organization? It would be extremely time-





consuming and a very monotonous task. Additionally, how confident would you be to repeat these tasks over and over without making a mistake that could impact production or the security of your infrastructure? That's where Infrastructure as Code (IaC) comes into the picture.

IaC allows us to easily replicate deployment tasks and take the human aspect out of repetitive tasks. By codifying your infrastructure, you can reduce or eliminate risks to the infrastructure running your applications. IaC makes changes idempotent, consistent, repeatable, and predictable. Plus, you can easily see how any modifications to your environment will impact your infrastructure before ever applying it.

Well, as much fun as that was, it's time to delete all of the resources that we just created. Note that normally you would need to delete the resources in a certain order since many of the resources are dependant on others. However, the VPC is unique in that you can delete the VPC and it will delete all associated resources if there aren't any other dependencies.

As you are deleting resources, note that manually deleting resources is sometimes risky, especially on a public cloud where forgotten resources could rack up a large bill. Additionally, leaving behind technical debt can impact future deployments and cause confusion when deploying future workloads.

## Task 6: Delete the VPC resources.

### Step 6.1

In the VPC Console, select the VPC that we just created by checking the tick box next to the VPC. From the Actions menu, select **Delete VPC**. Confirm you wish to delete the VPC and related AWS resources by typing *delete* in the text box at the bottom of the prompt. Click the **Delete** button.





## Hands-On Labs

The screenshot shows the AWS VPC Management console. On the left, there's a sidebar with navigation links like 'VPC Dashboard', 'Your VPCs', 'Route Tables', 'Internet Gateways', etc. The main area shows a table of VPCs with one row selected. A modal window titled 'Delete VPC' is open. It contains two sections: 'Will be deleted' (listing the VPC itself) and 'Will also be deleted' (listing five associated resources: demo-igw, public-rtb, public-subnet-1, public-subnet-2, and public-subnet-3). At the bottom of the modal, there's a text input field containing 'delete' with a red arrow labeled '1' pointing to it, and a large orange 'Delete' button with a red arrow labeled '2' pointing to it.

**Figure 17:** Create rtb

## Task 7: Prepare files and credentials for using Terraform to deploy cloud resources.

### Step 7.1

On your workstation, navigate to the `/workstation/terraform` directory. This is where we'll do all of our work for this training. Create a new file called `main.tf` and `variables.tf`.

In the `variables.tf`, copy the following variable definitions and save the file. Don't worry about understanding everything just yet, we'll learn all about variables in Objective 3.

```
variable "aws_region" {
  type    = string
  default = "us-east-1"
}
```





```

variable "vpc_name" {
  type    = string
  default = "demo_vpc"
}

variable "vpc_cidr" {
  type    = string
  default = "10.0.0.0/16"
}

variable "private_subnets" {
  default = {
    "private_subnet_1" = 1
    "private_subnet_2" = 2
    "private_subnet_3" = 3
  }
}

variable "public_subnets" {
  default = {
    "public_subnet_1" = 1
    "public_subnet_2" = 2
    "public_subnet_3" = 3
  }
}

```

In the `main.tf` file, copy the following Terraform configuration and save the file.

```

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

#Retrieve the list of AZs in the current AWS region
data "aws_availability_zones" "available" {}
data "aws_region" "current" {}

#Define the VPC
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
}

#Deploy the private subnets

```





```

resource "aws_subnet" "private_subnets" {
  for_each           = var.private_subnets
  vpc_id             = aws_vpc.vpc.id
  cidr_block         = cidrsubnet(var.vpc_cidr, 8, each.value)
  availability_zone = tolist(data.aws_availability_zones.available.names)[
    each.value]

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}

#Deploy the public subnets
resource "aws_subnet" "public_subnets" {
  for_each           = var.public_subnets
  vpc_id             = aws_vpc.vpc.id
  cidr_block         = cidrsubnet(var.vpc_cidr, 8, each.value + 100)
  availability_zone = tolist(data.aws_availability_zones.available.names)[each.value]
  map_public_ip_on_launch = true

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}

#Create route tables for public and private subnets
resource "aws_route_table" "public_route_table" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block     = "0.0.0.0/0"
    gateway_id     = aws_internet_gateway.internet_gateway.id
    #nat_gateway_id = aws_nat_gateway.nat_gateway.id
  }
  tags = {
    Name      = "demo_public_rtb"
    Terraform = "true"
  }
}

resource "aws_route_table" "private_route_table" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block     = "0.0.0.0/0"
    # gateway_id     = aws_internet_gateway.internet_gateway.id
}

```





```

    nat_gateway_id = aws_nat_gateway.nat_gateway.id
}
tags = {
  Name      = "demo_private_rtb"
  Terraform = "true"
}
}

#Create route table associations
resource "aws_route_table_association" "public" {
  depends_on      = [aws_subnet.public_subnets]
  route_table_id = aws_route_table.public_route_table.id
  for_each        = aws_subnet.public_subnets
  subnet_id       = each.value.id
}

resource "aws_route_table_association" "private" {
  depends_on      = [aws_subnet.private_subnets]
  route_table_id = aws_route_table.private_route_table.id
  for_each        = aws_subnet.private_subnets
  subnet_id       = each.value.id
}

#Create Internet Gateway
resource "aws_internet_gateway" "internet_gateway" {
  vpc_id = aws_vpc.vpc.id
  tags = {
    Name = "demo_igw"
  }
}

#Create EIP for NAT Gateway
resource "aws_eip" "nat_gateway_eip" {
  vpc      = true
  depends_on = [aws_internet_gateway.internet_gateway]
  tags = {
    Name = "demo_igw_eip"
  }
}

#Create NAT Gateway
resource "aws_nat_gateway" "nat_gateway" {
  depends_on      = [aws_subnet.public_subnets]
  allocation_id  = aws_eip.nat_gateway_eip.id
  subnet_id      = aws_subnet.public_subnets["public_subnet_1"].id
  tags = {
    Name = "demo_nat_gateway"
  }
}

```





## Task 8: Set credentials for Terraform deployment

### Step 8.1

Now that we have our Terraform files ready to go, the last step we need to complete is setting a few environment variables to set our AWS credentials and region used by Terraform. In AWS, generate an access key and secret key from an IAM user with Administrative privileges. If you need help, check out this link with a walk-through.

Once you have credentials, set the following environment variables for Linux, MacOS, or Bash on Windows:

```
export AWS_ACCESS_KEY_ID=""  
export AWS_SECRET_ACCESS_KEY=""
```

If you're running PowerShell on Windows, you'll need to use the following to set your AWS credentials:

```
PS C:\> $Env:AWS_ACCESS_KEY_ID=""  
PS C:\> $Env:AWS_SECRET_ACCESS_KEY=""
```

If you're using the default Windows command prompt, you can use the following to set your AWS credentials:

```
C:\> setx AWS_ACCESS_KEY_ID <YOUR ACCESS KEY>  
C:\> setx AWS_SECRET_ACCESS_KEY <YOUR SECRET KEY>
```

More information on setting credentials for AWS can be found [here](#)

## Task 9: Deploy the AWS infrastructure using Terraform

### Step 9.1

The first step to using Terraform is initializing the working directory. In your shell session, type the following command:

```
terraform init
```





As a response, you should see something like the following output (note that the provider version might change):

```
terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 3.37.0...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to
see
any changes that are required for your infrastructure. All Terraform
commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget,
other
commands will detect it and remind you to do so if necessary.
```

## Step 9.2

Now that our working directory is initialized, we can create a plan for execution. This will provide a preview of the changes to our AWS environment. To create a plan, execute the following command:

```
terraform plan
```

You should see an output similar to the one below. Note that the example below has been truncated for the sake of brevity:

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

data.aws_region.current: Refreshing state...
data.aws_availability_zones.available: Refreshing state...

-----
```

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create





Terraform will perform the following actions:

```
# aws_eip.nat_gateway_eip will be created
+ resource "aws_eip" "nat_gateway_eip" {
    + allocation_id      = (known after apply)
    + association_id    = (known after apply)
    + carrier_ip         = (known after apply)
    + customer_owned_ip = (known after apply)
    + domain             = (known after apply)
    + id                 = (known after apply)
    + instance            = (known after apply)
    + network_border_group = (known after apply)
    + network_interface   = (known after apply)
    + private_dns          = (known after apply)
    + private_ip           = (known after apply)
    + public_dns           = (known after apply)
    + public_ip             = (known after apply)
    + public_ipv4_pool     = (known after apply)
    + tags                = {
        + "Name" = "demo_igw_eip"
    }
    + vpc                 = true
}
...
}
```

Plan: 18 to add, 0 to change, 0 to destroy.

---

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

Notice that Terraform will create 18 resources for us in our AWS environment, which includes all of the networking components we manually created in earlier steps.

### Step 9.3

For our final step to create our AWS resources, we need to apply the configuration. An apply will instruct Terraform to create the resources in AWS that are defined in our configuration file(s). And as we saw in our plan, it will create 18 resources for us. To execute the Terraform, run the following command:

```
terraform apply -auto-approve
```





Note that we are using the `-auto-approve` flag for simplicity. You can leave it out, validate the changes to the environment, and type `yes` to validate you want to apply the configuration.

After running the above command, you should see output similar to the following:

```
data.aws_region.current: Refreshing state...
data.aws_availability_zones.available: Refreshing state...

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_eip.nat_gateway_eip will be created
+ resource "aws_eip" "nat_gateway_eip" {
    + allocation_id      = (known after apply)
    + association_id    = (known after apply)
    + carrier_ip         = (known after apply)
    + customer_owned_ip = (known after apply)
    + domain             = (known after apply)
    + id                 = (known after apply)
    + instance            = (known after apply)
    + network_border_group = (known after apply)
    + network_interface   = (known after apply)
    + private_dns          = (known after apply)
    + private_ip           = (known after apply)
    + public_dns           = (known after apply)
    + public_ip             = (known after apply)
    + public_ipv4_pool     = (known after apply)
    + tags                = {
        + "Name" = "demo_igw_eip"
    }
    + vpc                 = true
}

...
aws_vpc.vpc: Creating...
aws_vpc.vpc: Still creating... [10s elapsed]
aws_vpc.vpc: Creation complete after 11s [id=vpc-05bf9e6a14d8ee736]
aws_subnet.private_subnets["private_subnet_2"]: Creating...
aws_subnet.public_subnets["public_subnet_3"]: Creating...
aws_internet_gateway.internet_gateway: Creating...
aws_subnet.public_subnets["public_subnet_1"]: Creating...
aws_subnet.private_subnets["private_subnet_1"]: Creating...
aws_subnet.public_subnets["public_subnet_2"]: Creating...
aws_subnet.private_subnets["private_subnet_3"]: Creating...
aws_subnet.private_subnets["private_subnet_3"]: Creation complete after 2s
[id=subnet-097ce13a4cd397b92]
```





```

aws_subnet.private_subnets["private_subnet_2"]: Creation complete after 2s
  [id=subnet-0e13dbad1bdcc9b3d]
aws_internet_gateway.internet_gateway: Creation complete after 3s [id=igw-09460c69ff2efdaa7]
aws_eip.nat_gateway_eip: Creating...
aws_route_table.public_route_table: Creating...
aws_subnet.private_subnets["private_subnet_1"]: Creation complete after 3s
  [id=subnet-0cd287fb292ad3720]
aws_eip.nat_gateway_eip: Creation complete after 1s [id=eipalloc-0834ec4c7cb0ff0d1]
aws_route_table.public_route_table: Creation complete after 2s [id=rtb-0f354a5a7facfe6f1]
aws_subnet.public_subnets["public_subnet_1"]: Still creating... [10s elapsed]
aws_subnet.public_subnets["public_subnet_2"]: Still creating... [10s elapsed]
aws_subnet.public_subnets["public_subnet_3"]: Still creating... [10s elapsed]
aws_subnet.public_subnets["public_subnet_3"]: Creation complete after 13s
  [id=subnet-09b2418caea54512e]
aws_subnet.public_subnets["public_subnet_2"]: Creation complete after 14s
  [id=subnet-0fd89ea15770c4658]
aws_subnet.public_subnets["public_subnet_1"]: Creation complete after 15s
  [id=subnet-0a1cc28bb3bb0d318]
aws_nat_gateway.nat_gateway: Creating...
aws_route_table_association.public["public_subnet_3"]: Creating...
aws_route_table_association.public["public_subnet_1"]: Creating...
aws_route_table_association.public["public_subnet_2"]: Creating...
aws_route_table_association.public["public_subnet_2"]: Creation complete
  after 0s [id=rtbassoc-014bdb5bfb698ad04]
aws_route_table_association.public["public_subnet_3"]: Creation complete
  after 0s [id=rtbassoc-0bc228d98326cbfac]
aws_route_table_association.public["public_subnet_1"]: Creation complete
  after 0s [id=rtbassoc-0b9c0198f2dbac2c9]
aws_nat_gateway.nat_gateway: Still creating... [10s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [20s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [30s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [40s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [50s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [1m0s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [1m10s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [1m20s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [1m30s elapsed]
aws_nat_gateway.nat_gateway: Creation complete after 1m35s [id=nat-037dd8aec387c4069]
aws_route_table.private_route_table: Creating...
aws_route_table.private_route_table: Creation complete after 3s [id=rtb-0d564cf66b5ca1136]
aws_route_table_association.private["private_subnet_3"]: Creating...

```





```

aws_route_table_association.private["private_subnet_1"]: Creating...
aws_route_table_association.private["private_subnet_2"]: Creating...
aws_route_table_association.private["private_subnet_3"]: Creation complete
  after 1s [id=rtbassoc-0f593a3305dd07803]
aws_route_table_association.private["private_subnet_2"]: Creation complete
  after 1s [id=rtbassoc-0e0c79dd4d1bd7dc7]
aws_route_table_association.private["private_subnet_1"]: Creation complete
  after 2s [id=rtbassoc-0a3019304223a483a]

Apply complete! Resources: 18 added, 0 changed, 0 destroyed.

```

At this point, Terraform has created new resources in our AWS account that match the requirements stated at the beginning of the lab. Feel free to log into the AWS console and browse around. You should see the new VPC, subnets, route tables, NAT Gateway, and Internet Gateway. These should look just like our configuration, but completely automated by Terraform.

## Task 10: Delete the AWS resources using Terraform to clean up our AWS environment

### Step 10.1

The final step is to destroy all of the resources created by Terraform. By using Terraform to destroy the resources, you will ensure that every single resource deployed with Terraform is destroyed from your account. This ensures you don't leave anything behind that could incur costs or leave behind technical debt.

To destroy your resources, execute the following command in the terminal. You should see Terraform refresh the state of each resource and subsequently destroy it in the proper order.

```
terraform destroy -auto-approve
```

You should see an output similar to the following:

```

aws_nat_gateway.nat_gateway: Destroying... [id=nat-037dd8aec387c4069]
aws_route_table.public_route_table: Destruction complete after 3s
aws_nat_gateway.nat_gateway: Still destroying... [id=nat-037dd8aec387c4069
  , 10s elapsed]
aws_nat_gateway.nat_gateway: Still destroying... [id=nat-037dd8aec387c4069
  , 20s elapsed]
aws_nat_gateway.nat_gateway: Still destroying... [id=nat-037dd8aec387c4069
  , 30s elapsed]
aws_nat_gateway.nat_gateway: Still destroying... [id=nat-037dd8aec387c4069
  , 40s elapsed]
aws_nat_gateway.nat_gateway: Still destroying... [id=nat-037dd8aec387c4069
  , 50s elapsed]

```





```
aws_nat_gateway.nat_gateway: Destruction complete after 56s
aws_subnet.public_subnets["public_subnet_3"]: Destroying... [id=subnet-09
    b2418caeaa54512e]
aws_eip.nat_gateway_eip: Destroying... [id=eipalloc-0834ec4c7cb0ff0d1]
aws_subnet.public_subnets["public_subnet_2"]: Destroying... [id=subnet-0
    fd89ea15770c4658]
aws_subnet.public_subnets["public_subnet_1"]: Destroying... [id=subnet-0
    a1cc28bb3bb0d318]
aws_subnet.public_subnets["public_subnet_2"]: Destruction complete after 1
    s
aws_subnet.public_subnets["public_subnet_3"]: Destruction complete after 1
    s
aws_subnet.public_subnets["public_subnet_1"]: Destruction complete after 1
    s
aws_eip.nat_gateway_eip: Destruction complete after 1s
aws_internet_gateway.internet_gateway: Destroying... [id=igw-09460
    c69ff2efdaa7]
aws_internet_gateway.internet_gateway: Still destroying... [id=igw-09460
    c69ff2efdaa7, 10s elapsed]
aws_internet_gateway.internet_gateway: Destruction complete after 12s
aws_vpc.vpc: Destroying... [id=vpc-05bf9e6a14d8ee736]
aws_vpc.vpc: Destruction complete after 1s

Destroy complete! Resources: 18 destroyed.
```

**Congratulations, you've reached the end of this lab.**

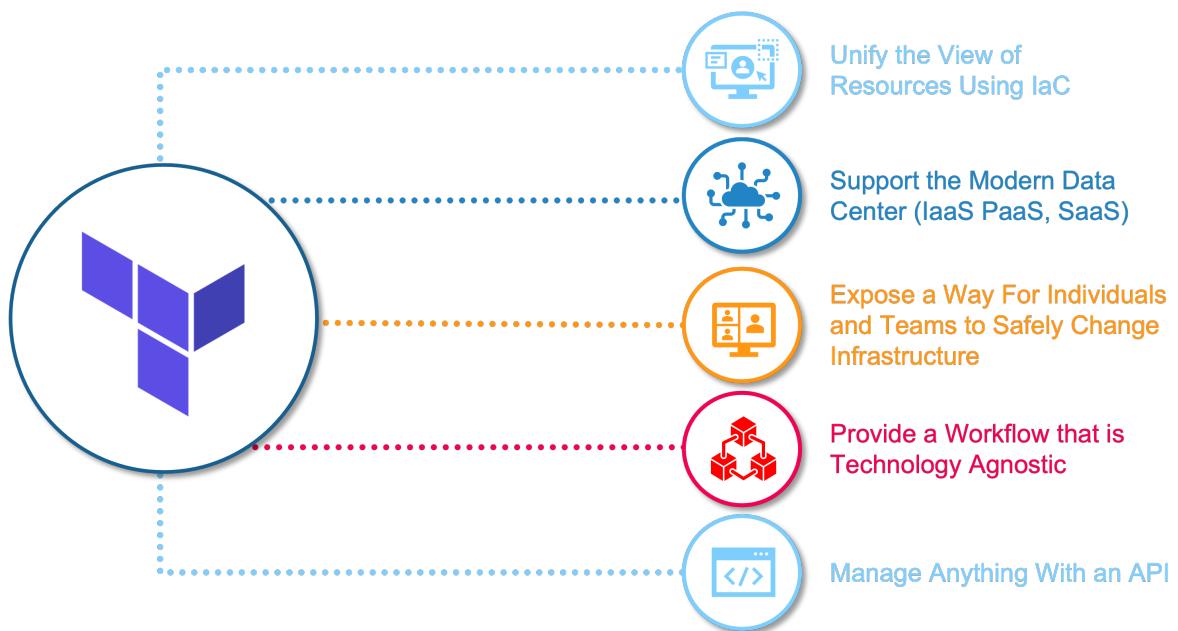




## Lab: Terraform's Purpose

### Terraform Goals

- Unify the view of resources using infrastructure as code
- Support the modern data center (IaaS, PaaS, SaaS)
- Expose a way for individuals and teams to safely and predictably change infrastructure
- Provide a workflow that is technology agnostic
- Manage anything with an API



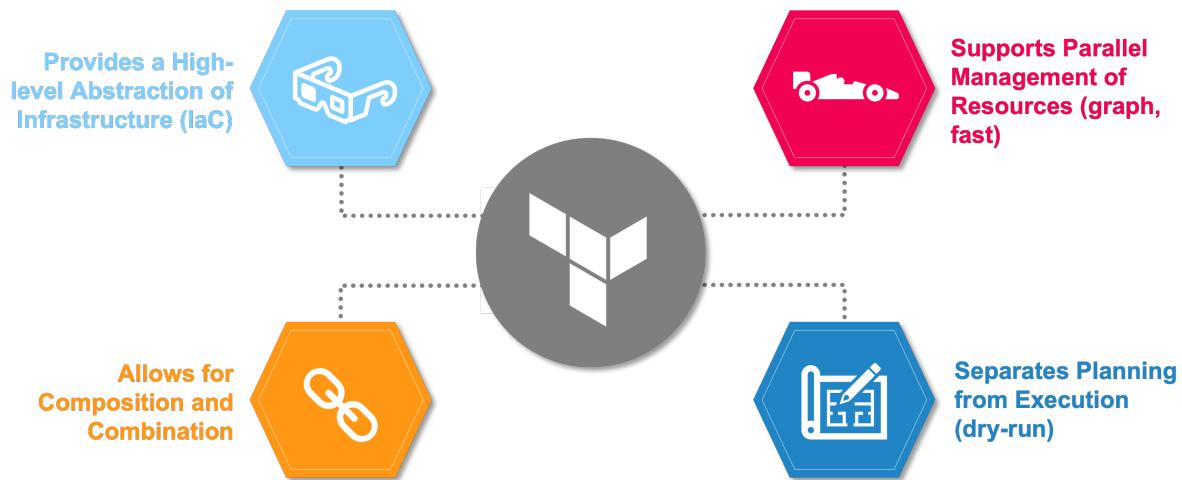
### Terraform Benefits

- Provides a high-level abstraction of infrastructure (IaC)
- Allows for composition and combination
- Supports parallel management of resources (graph, fast)
- Separates planning from execution (dry-run)





Hands-On Labs





## Lab: Benefits of State

### Terraform State

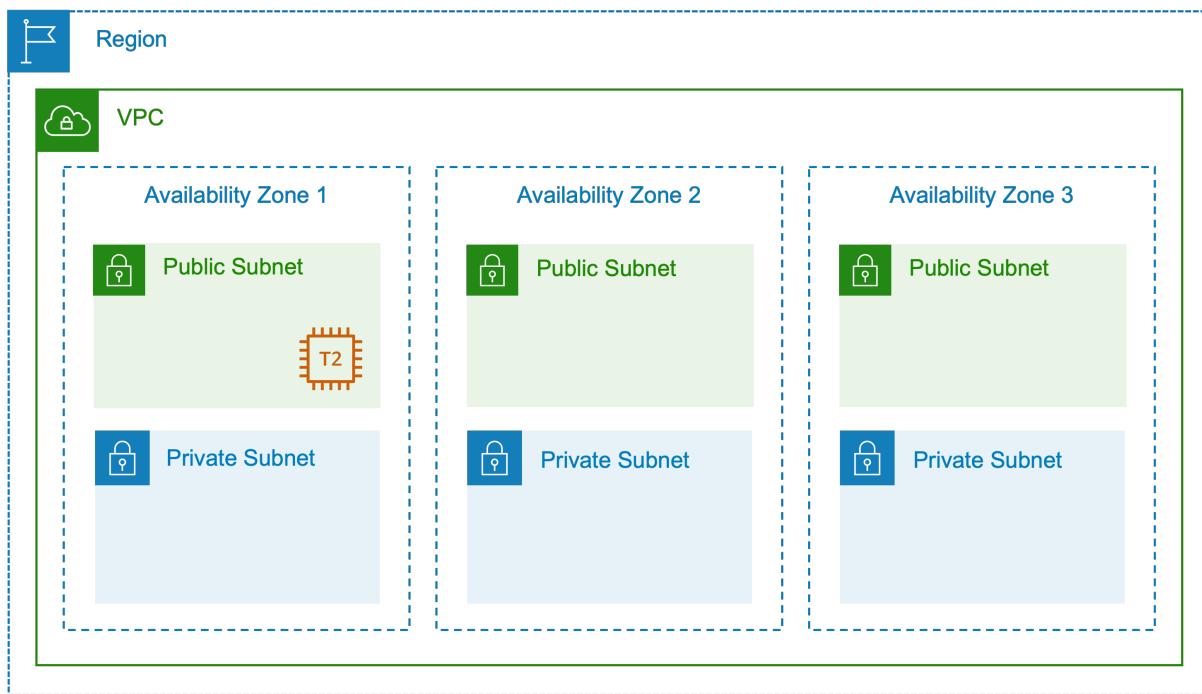
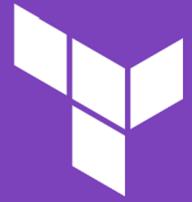
In order to properly and correctly manage your infrastructure resources, Terraform stores the state of your managed infrastructure. Terraform uses this state on each execution to plan and make changes to your infrastructure. This state must be stored and maintained on each execution so future operations can perform correctly.

### Benefits of State

During execution, Terraform will examine the state of the currently running infrastructure, determine what differences exist between the current state and the revised desired state, and indicate the necessary changes that must be applied. When approved to proceed, only the necessary changes will be applied, leaving existing, valid infrastructure untouched.

- Task 1: Show Current State
- Task 2: Update your Configuration
- Task 3: Plan and Execute Changes
- Task 4: Show New State





**Figure 1:** AWS Application Infrastructure Buildout

### Task 1: Show Current State

In a previous lab, you wrote some Terraform configuration using the HashiCorp Configuration Language to create a new VPC in AWS. Now that that VPC exists we will build an EC2 instance in one of the public subnets. Since the VPC still exists the only change that Terraform needs to address is the addition of the EC2 instance.

#### Step 1.1.1

On your workstation, navigate to the `/workstation/terraform` directory. To view the applied configuration utilize the `terraform show` command to view the resources created and find the IP address for your instance.

Note: If this command doesn't yield any information then you will need to redeploy your VPC infrastructure following the steps in Objective 1b. This directory should contain both a `main.tf` and `variables.tf` file.





```
terraform show
```

```
terraform show
```

```
# aws_eip.nat_gateway_eip:
resource "aws_eip" "nat_gateway_eip" {
    association_id      = "eipassoc-0f2986cc722254f2e"
    domain              = "vpc"
    id                  = "eipalloc-000eb0775a52a1e32"
    network_border_group = "us-east-1"
    network_interface   = "eni-0272f17827cbb823a"
    private_dns         = "ip-10-0-101-134.ec2.internal"
    private_ip          = "10.0.101.134"
    public_dns          = "ec2-54-205-175-114.compute-1.amazonaws.com"
    public_ip           = "54.205.175.114"
    public_ipv4_pool    = "amazon"
    tags                = {
        "Name" = "demo_igw_eip"
    }
    tags_all            = {
        "Name" = "demo_igw_eip"
    }
    vpc                = true
}

# aws_internet_gateway.internet_gateway:
resource "aws_internet_gateway" "internet_gateway" {
    arn      = "arn:aws:ec2:us-east-1:xxx:internet-gateway/igw-0bexxx"
    id       = "igw-0be99153cf7f3c6ab"
    owner_id = "508140242758"
    tags     = {
        "Name" = "demo_igw"
    }
    tags_all = {
        "Name" = "demo_igw"
    }
    vpc_id   = "vpc-064a97911d85e16d4"
}

# aws_nat_gateway.nat_gateway:
resource "aws_nat_gateway" "nat_gateway" {
    allocation_id      = "eipalloc-000eb0775a52a1e32"
    connectivity_type  = "public"
    id                = "nat-0705f0f6101212b3b"
    network_interface_id = "eni-0272f17827cbb823a"
    private_ip         = "10.0.101.134"
    public_ip          = "54.205.175.114"
```





...

## Task 2: Update your Configuration to include EC2 instance

Terraform can perform in-place updates after changes are made to the `main.tf` configuration file. Update your `main.tf` to include an EC2 instance in the public subnet:

Append the following code to `main.tf`

```
# Terraform Data Block - To Lookup Latest Ubuntu 20.04 AMI Image
data "aws_ami" "ubuntu" {
    most_recent = true

    filter {
        name    = "name"
        values  = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
    }

    filter {
        name    = "virtualization-type"
        values  = ["hvm"]
    }

    owners = ["099720109477"]
}

# Terraform Resource Block - To Build EC2 instance in Public Subnet
resource "aws_instance" "web_server" {
    ami          = data.aws_ami.ubuntu.id
    instance_type = "t2.micro"
    subnet_id     = aws_subnet.public_subnets["public_subnet_1"].id
    tags = {
        Name = "Ubuntu EC2 Server"
    }
}
```

Save the configuration.

## Task 3: Plan and Execute Changes

Plan and apply the changes you just made and note the output differences for additions, deletions, and in-place changes.





### Step 3.1.1

Run a `terraform plan` to see the updates that Terraform needs to make.

```
terraform plan
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

- + create

Terraform will perform the following actions:

```
# aws_instance.web_server will be created
+ resource "aws_instance" "web_server" {
    + ami                                = "ami-036490d46656c4818"
    + arn                                = (known after apply)
    + associate_public_ip_address        = (known after apply)
    + availability_zone                  = (known after apply)
    + cpu_core_count                     = (known after apply)
    + cpu_threads_per_core              = (known after apply)
    + disable_api_termination           = (known after apply)
    + ebs_optimized                      = (known after apply)
    + get_password_data                 = false
    + host_id                            = (known after apply)
    + id                                 = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_state                     = (known after apply)
    + instance_type                      = "t2.micro"
  ...
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

### Step 3.1.2

Run a `terraform apply` to execute the updates that Terraform needs to make.

```
terraform apply
```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value:





When prompted to apply the changes, respond with `yes`.

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

aws_instance.web_server: Creating...
aws_instance.web_server: Still creating... [10s elapsed]
aws_instance.web_server: Still creating... [20s elapsed]
aws_instance.web_server: Still creating... [30s elapsed]
aws_instance.web_server: Still creating... [40s elapsed]
aws_instance.web_server: Still creating... [50s elapsed]
aws_instance.web_server: Creation complete after 59s [id=i-0d544e90777ca8c2f]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

### Task 3: Show New State

To view the applied configuration utilize the `terraform show` command to view the resources created. Look for the `aws_instance.web_server` which is now present within Terraform's managed state.

```
terraform show
```

Terraform State example:

```
# aws_instance.web_server:
resource "aws_instance" "web_server" {
    ami
    arn
    -1:508140242758:instance/i-0d544e90777ca8c2f"
    associate_public_ip_address
    availability_zone
    cpu_core_count
    cpu_threads_per_core
    disable_api_termination
    ebs_optimized
    get_password_data
    hibernation
    id
    instance_initiated_shutdown_behavior
    instance_state
    instance_type
    ipv6_address_count
    = "ami-036490d46656c4818"
    = "arn:aws:ec2:us-east-1"
    = "true"
    = "us-east-1b"
    = 1
    = 1
    = false
    = false
    = false
    = false
    = "i-0d544e90777ca8c2f"
    = "stop"
    = "running"
    = "t2.micro"
    = 0
```





## Hands-On Labs

```

  ipv6_addresses
  monitoring
  primary_network_interface_id
  private_dns
  private_ip
  public_ip
  secondary_private_ips
  security_groups
  source_dest_check
  subnet_id
  tags
    "Name" = "Ubuntu EC2 Server"
  }
  tags_all
    "Name" = "Ubuntu EC2 Server"
  }
  tenancy
  vpc_security_group_ids
    "sg-097b59a05720fb97c",
  ]

capacity_reservation_specification {
  capacity_reservation_preference = "open"
}

credit_specification {
  cpu_credits = "standard"
}

enclave_options {
  enabled = false
}

metadata_options {
  http_endpoint          = "enabled"
  http_put_response_hop_limit = 1
  http_tokens            = "optional"
}

root_block_device {
  delete_on_termination = true
  device_name           = "/dev/sda1"
  encrypted              = false
  iops                  = 100
  tags                  = {}
  throughput             = 0
  volume_id              = "vol-053758fb913734c4c"
  volume_size             = 8
  volume_type             = "gp2"
}

```





```
}
```

Alternatively you can run a `terraform state list` to list all of the items in Terraform's managed state.

```
terraform state list
```

```
data.aws_ami.ubuntu
data.aws_availability_zones.available
data.aws_region.current
aws_eip.nat_gateway_eip
aws_instance.web_server
aws_internet_gateway.internet_gateway
aws_nat_gateway.nat_gateway
aws_route_table.private_route_table
aws_route_table.public_route_table
aws_route_table_association.private["private_subnet_1"]
aws_route_table_association.private["private_subnet_2"]
aws_route_table_association.private["private_subnet_3"]
aws_route_table_association.public["public_subnet_1"]
aws_route_table_association.public["public_subnet_2"]
aws_route_table_association.public["public_subnet_3"]
aws_subnet.private_subnets["private_subnet_1"]
aws_subnet.private_subnets["private_subnet_2"]
aws_subnet.private_subnets["private_subnet_3"]
aws_subnet.public_subnets["public_subnet_1"]
aws_subnet.public_subnets["public_subnet_2"]
aws_subnet.public_subnets["public_subnet_3"]
aws_vpc.vpc
```





## Lab: Terraform Basics

All interactions with Terraform occur via the CLI. Terraform is a local tool (runs on the current machine). The terraform ecosystem also includes providers for many cloud services, and a module repository. Hashicorp also has products to help teams manage Terraform: Terraform Cloud and Terraform Enterprise.

There are a handful of basic terraform commands, including:

- `terraform init`
- `terraform validate`
- `terraform plan`
- `terraform apply`
- `terraform destroy`

These commands make up the terraform workflow that we will cover in objective 6 of this course. It will be beneficial for us to explore some basic commands now so that work alongside and deploy our configurations.

- Task 1: Verify Terraform installation and version
- Task 2: Initialize Terraform Working Directory: `terraform init`
- Task 3: Validating a Configuration: `terraform validate`
- Task 4: Generating a Terraform Plan: `terraform plan`
- Task 5: Applying a Terraform Plan: `terraform apply`
- Task 6: Terraform Destroy: `terraform destroy`

### Task 1: Verify Terraform installation and version

You can get the version of Terraform running on your machine with the following command:

```
terraform -version
```

If you need to recall a specific subcommand, you can get a list of available commands and arguments with the `help` argument.

```
terraform -help
```





## Task 2: Terraform Init

Initializing your workspace is used to initialize a working directory containing Terraform configuration files.

Copy the code snippet below into the file called `main.tf`. This snippet leverages the random provider, maintained by HashiCorp, to generate a random string.

```
main.tf
```

```
resource "random_string" "random" {
  length = 16
}
```

Once saved, you can return to your shell and run the `init` command shown below. This tells Terraform to scan your code and download anything it needs locally.

```
terraform init
```

Once your Terraform workspace has been initialized you are ready to begin planning and provisioning your resources.

Note: You can validate that your workspace is initialized by looking for the presence of a `.terraform` directory. This is a hidden directory, which Terraform uses to manage cached provider plugins and modules, record which workspace is currently active, and record the last known backend configuration in case it needs to migrate state. This directory is automatically managed by Terraform, and is created during initialization.

## Task 3: Validating a Configuration

The `terraform validate` command validates the configuration files in your working directory.

To validate there are no syntax problems with our terraform configuration file run a

```
terraform validate
```

```
Success! The configuration is valid.
```

## Task 4: Generating a Terraform Plan

Terraform has a dry run mode where you can preview what Terraform will change without making any actual changes to your infrastructure. This dry run is performed by running a `terraform plan`.





In your terminal, you can run a plan as shown below to see the changes required for Terraform to reach your desired state you defined in your code. This is equivalent to running Terraform in a “dry” mode.

```
terraform plan
```

If you review the output, you will see 1 change will be made which is to generate a single random string.

Terraform will perform the following actions:

```
# random_string.random will be created
+ resource "random_string" "random" {
    + id          = (known after apply)
    + length      = 16
    + lower       = true
    + min_lower   = 0
    + min_numeric = 0
    + min_special = 0
    + min_upper   = 0
    + number      = true
    + result      = (known after apply)
    + special     = true
    + upper       = true
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Note: Terraform also has the concept of planning out changes to a file. This is useful to ensure you only apply what has been planned previously. Try running a plan again but this time passing an -out flag as shown below.

```
terraform plan -out myplan
```

This will create a plan file that Terraform can use during an `apply`.

## Task 5: Applying a Terraform Plan

Run the command below to build the resources within your plan file.

```
terraform apply myplan
```

Once completed, you will see Terraform has successfully built your random string resource based on what was in your plan file.





## Hands-On Labs

Terraform can also run an `apply` without a plan file. To try it out, modify your `main.tf` file to create a random string with a length of 10 instead of 16 as shown below:

```
resource "random_string" "random" {  
    length = 10  
}
```

and run a `terraform apply`

```
terraform apply
```

Notice you will now see a similar output to when you ran a `terraform plan` but you will now be asked if you would like to proceed with those changes. To proceed enter `yes`.

```
Terraform used the selected providers to generate the following execution  
plan. Resource actions are indicated with the  
following symbols:  
-/+ destroy and then create replacement
```

Terraform will perform the following actions:

```
# random_string.random must be replaced  
-/+ resource "random_string" "random" {  
    ~ id          = "XW>5m{w8Ig96d1A&" -> (known after apply)  
    ~ length      = 16 -> 10 # forces replacement  
    ~ result      = "XW>5m{w8Ig96d1A&" -> (known after apply)  
    # (8 unchanged attributes hidden)  
}
```

Plan: 1 to add, 0 to change, 1 to destroy.

```
Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.
```

Enter a value:

Once complete the random string resource will be created with the attributes specified in the `main.tf` configuration file.

### Task 6: Terraform Destroy

The `terraform destroy` command is a convenient way to destroy all remote objects managed by a particular Terraform configuration. It does not delete your configuration file(s), `main.tf`, etc. It destroys the resources built from your Terraform code.





Run the command as shown below to run a planned destroy:

```
terraform plan -destroy
```

Terraform will perform the following actions:

```
# random_string.random will be destroyed
- resource "random_string" "random" {
    - id          = "1HIQs)moC0" -> null
    - length      = 10 -> null
    - lower       = true -> null
    - min_lower   = 0 -> null
    - min_numeric = 0 -> null
    - min_special = 0 -> null
    - min_upper   = 0 -> null
    - number      = true -> null
    - result      = "1HIQs)moC0" -> null
    - special     = true -> null
    - upper       = true -> null
}
```

Plan: 0 to add, 0 to change, 1 to destroy.

You will notice that it is planning to destroy your previously created resource. To actually destroy the random string you created, you can run a destroy command as shown below.

```
terraform destroy
```

Terraform will perform the following actions:

```
# random_string.random will be destroyed
- resource "random_string" "random" {
    - id          = "1HIQs)moC0" -> null
    - length      = 10 -> null
    - lower       = true -> null
    - min_lower   = 0 -> null
    - min_numeric = 0 -> null
    - min_special = 0 -> null
    - min_upper   = 0 -> null
    - number      = true -> null
    - result      = "1HIQs)moC0" -> null
    - special     = true -> null
    - upper       = true -> null
}
```

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?





Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only '`yes`' will be accepted to confirm.

Enter a value:

Note: As similar to when you ran an apply, you will be prompted to proceed with the destroy by entering "yes".





## Lab: HashiCorp Configuration Language

## Hands-On Labs

Terraform is written in HCL (HashiCorp Configuration Language) and is designed to be both human and machine readable. HCL is built using code configuration blocks which typically follow the following syntax:

```
1 # Template
2 <BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
3     # Block body
4     <IDENTIFIER> = <EXPRESSION> # Argument
5 }
6
7 # AWS EC2 Example
8 resource "aws_instance" "web_server" { # BLOCK
9     ami           = "ami-04d29b6f966df1537" # Argument
10    instance_type = var.instance_type # Argument with value as expression (Variable value r
11 }
```

Terraform Code Configuration block types include:

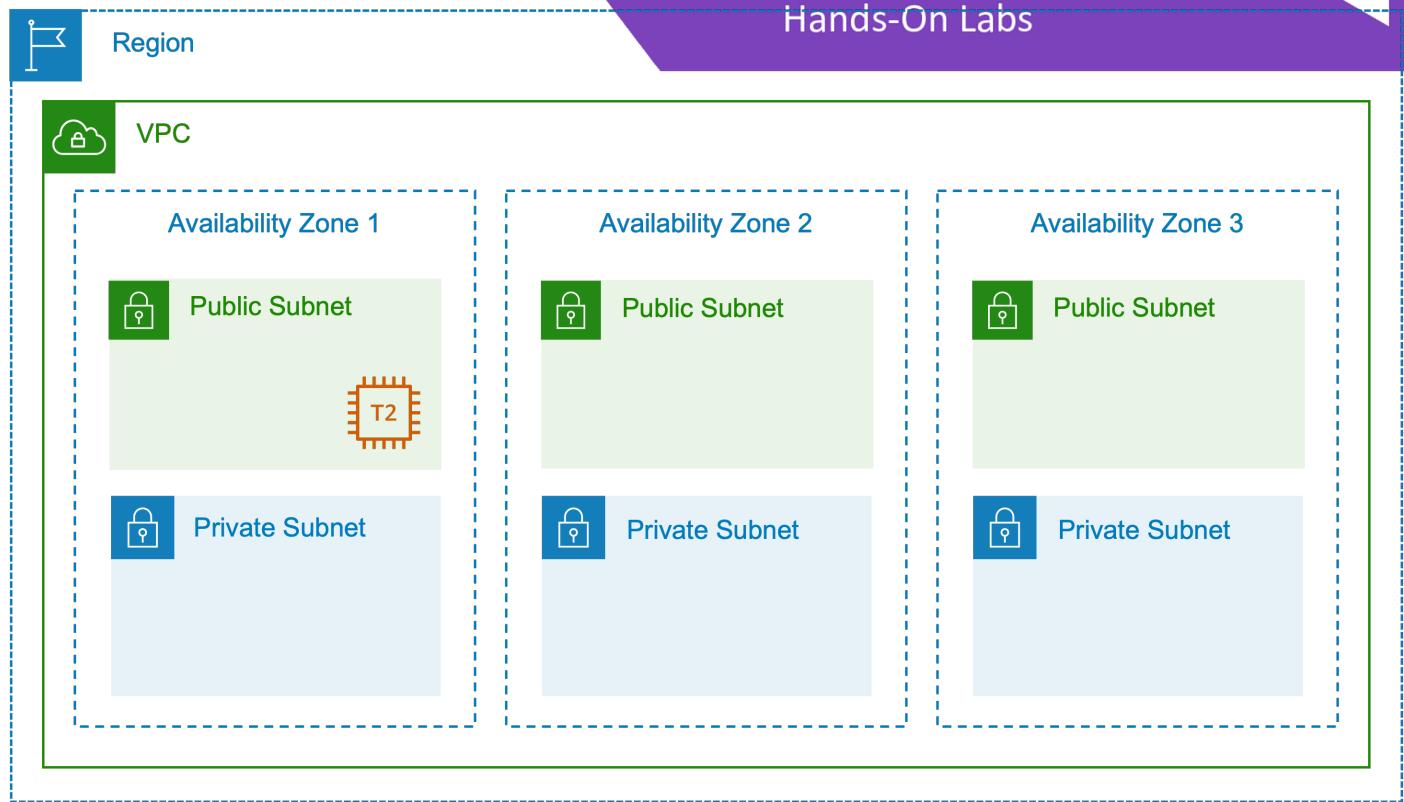
- Terraform Settings Block
- Terraform Provider Block
- Terraform Resource Block
- Terraform Data Block
- Terraform Input Variables Block
- Terraform Local Variables Block
- Terraform Output Values Block
- Terraform Modules Block

We will be utilizing Terraform Provider, Terraform Resource, Data and Input Variables Blocks in this lab. This course will go through each of these configuration blocks in more detail throughout the course.

- Task 1: Connect to the Student Workstation
- Task 2: Verify Terraform installation
- Task 3: Update Terraform Configuration to include EC2 instance
- Task 4: Use the Terraform CLI to Get Help
- Task 5: Apply your Configuration
- Task 6: Verify EC2 Server in AWS Management Console

Created by Gabe Maentz and Bryan Krausen





**Figure 1:** AWS Application Infrastructure Buildout

### Task 1: Connect to the Student Workstation

In the previous lab, you learned how to connect to your workstation with either VSCode, SSH, or the web-based client.

Once you've connected, make sure you've navigated to the `/workstation/terraform` directory. This is where we'll do all of our work for this training.

### Task 2: Verify Terraform installation

#### Step 1.2.1

Run the following command to check the Terraform version:

```
1 terraform -version
```

You should see:

```
1 Terraform v1.0.8
```

Created by Gabe Maentz and Bryan Krausen





## Hands On Labs

### Task 3: Update Terraform Configuration to include EC2 instance

#### Step 1.3.1

In the /workstation/terraform directory, edit the file titled `main.tf` to create an AWS EC2 instance within one of the our public subnets.

Your final `main.tf` file should look similar to this with different values:

```
1 provider "aws" {
2   access_key = "<YOUR_ACCESSKEY>"
3   secret_key = "<YOUR_SECRETKEY>"
4   region     = "<REGION>"
5 }
6
7 resource "aws_instance" "web" {
8   ami           = "<AMI>"
9   instance_type = "t2.micro"
10
11  subnet_id        = "<SUBNET>"
12  vpc_security_group_ids = ["<SECURITY_GROUP>"]
13
14  tags = {
15    "Identity" = "<IDENTITY>"
16  }
17 }
```

Don't forget to save the file before moving on!

### Task 4: Use the Terraform CLI to Get Help

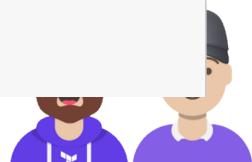
#### Step 1.4.1

Execute the following command to display available commands:

```
1 terraform -help
```

```
1 Usage: terraform [-version] [-help] <command> [args]
2
3 The available commands for execution are listed below.
4 The most common, useful commands are shown first, followed by
5 less common or more advanced commands. If you're just getting
6 started with Terraform, stick with the common commands. For the
7 other commands, please read the help and docs before usage.
8
9 Common commands:
10 apply          Builds or changes infrastructure
11 console        Interactive console for Terraform interpolations
12 destroy        Destroy Terraform-managed infrastructure
13 env            Workspace management
14 fmt            Rewrites config files to canonical format
```

Created by Gabe Maentz and Bryan Krausen





Terraform Commands		Usage Examples
15      get		Download and install modules for the configuration
16      graph		Create a visual graph of Terraform resources
17      import		Import existing infrastructure into Terraform
18      init		Initialize a Terraform working directory
19      output		Read an output from a state file
20      plan		Generate and show an execution plan
21		
22      ...		

- (full output truncated for sake of brevity in this guide)

Or, you can use short-hand:

```
1 terraform -h
```

### Step 1.4.2

Navigate to the Terraform directory and initialize Terraform

```
1 cd /workstation/terraform
```

```
1 terraform init
```

```
1 Initializing provider plugins...
2 ...
3
4 Terraform has been successfully initialized!
```

### Step 1.4.3

Get help on the `plan` command and then run it:

```
1 terraform -h plan
```

```
1 terraform plan
```

## Task 5: Apply your Configuration

### Step 1.5.1

Run the `terraform apply` command to generate real resources in AWS

```
1 terraform apply
```

You will be prompted to confirm the changes before they're applied. Respond with `yes`.

Created by Gabe Maentz and Bryan Krausen





## Task 6: Verify EC2 Server in AWS Management Console Hands-On Labs

Login to AWS Management Console -> Services -> EC2 to verify newly created EC2 instance

Instance ID	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4
i-040805d3f31a4ea4f	Terraform HOL Server	i-040805d3f31a4ea4f	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	-

**Instance: i-040805d3f31a4ea4f (Terraform HOL Server)**

Details			Security			Networking			Storage			Status checks			Monitoring			Tags		
<b>Instance summary</b> <a href="#">Info</a>																				
Instance ID	i-040805d3f31a4ea4f (Terraform HOL Server)		Public IPv4 address	<a href="#">34.201.205.45   open address</a>		Private IPv4 addresses			Private IPv4 addresses	<a href="#">10.0.101.171</a>		Public IPv4 DNS			Public IPv4 DNS			IAM Role		
IPv6 address	-		Instance state	<a href="#">Running</a>		Instance type	t2.micro		AWS Compute Optimizer finding	<a href="#">Opt-in to AWS Compute Optimizer for recommendations.   Learn more</a>		Elastic IP addresses			Elastic IP addresses					
Private IPv4 DNS	<a href="#">ip-10-0-101-171.ec2.internal</a>		VPC ID	<a href="#">vpc-029c8f08991bad7fe (demo_vpc)</a>		Subnet ID	<a href="#">subnet-0698af9fe82d2cf7e (public_subnet_1)</a>		Platform	AMI ID		AMI name			Monitoring	disabled		Termination protection	Disabled	
Platform	<a href="#">Ubuntu (Inferred)</a>		Platform details	<a href="#">Linux/UNIX</a>		AMI ID	<a href="#">ami-036490d46656c4818</a>		AMI name	<a href="#">ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20211001</a>		Monitoring	disabled		Termination protection	Disabled				

**Figure 2:** AWS EC2 Server

## References

Terraform Configuration Terraform Configuraiton Syntax



Created by Gabe Maentz and Bryan Krausen



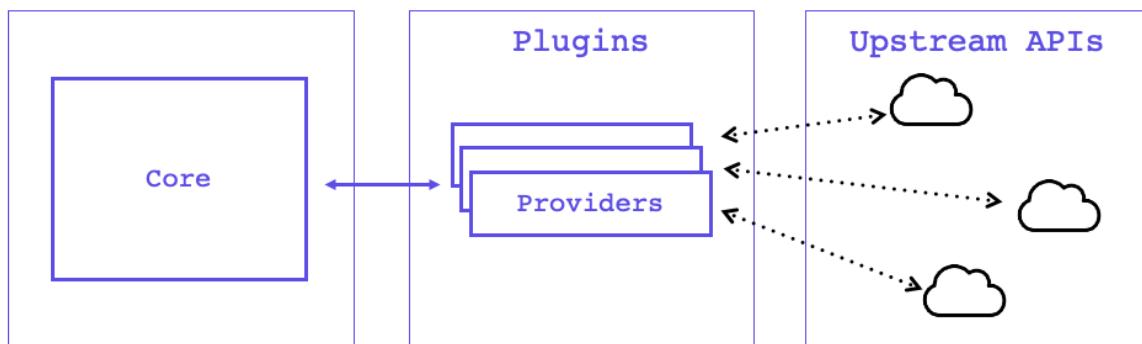
## Lab: Terraform Plug-in Based Architecture

Terraform relies on plugins called “providers” to interact with remote systems and expand functionality. Terraform configurations must declare which providers they require so that Terraform can install and use them. This is performed within a Terraform configuration block.

- Task 1: View available Terraform Providers
- Task 2: Install the Terraform AWS Provider
- Task 3: View installed and required providers

### Task 1: View available Terraform Providers

Terraform Providers are plugins that implement resource types for particular clouds, platforms and generally speaking any remote system with an API. Terraform configurations must declare which providers they require, so that Terraform can install and use them. Popular Terraform Providers include: AWS, Azure, Google Cloud, VMware, Kubernetes and Oracle.



**Figure 1:** Terraform Plug-in Architecture

For a full list of available Terraform providers, reference the [Terraform Provider Registry](#)



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

A screenshot of the HashiCorp Terraform Registry website. The top navigation bar includes the HashiCorp logo, 'Terraform Registry', a search bar ('Search Providers and Modules'), and links for 'Browse', 'Publish', and 'Sign-in'. Below the navigation is a menu with 'Providers' (selected) and 'Modules'. On the left, there are 'FILTERS' (with 'Official', 'Verified', and 'Community' checked), a 'Category' sidebar listing various provider types like HashiCorp Platform, Public Cloud, Asset Management, etc., and a footer link 'try.terraform.io'. The main content area is titled '# Providers' and contains a brief description: 'Providers are a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources.' Below this are six provider cards arranged in two rows of three: AWS (orange background, AWS logo), Azure (blue background, Azure logo), Google Cloud Platform (blue background, GCP logo), Kubernetes (blue background, Kubernetes logo), Oracle Cloud Infrastructure (red background, Oracle logo), and Alibaba Cloud (dark grey background, Alibaba logo). At the bottom of the page are links for Active Directory, Archive, AWS Cloud Control, and a 'try.terraform.io' button.

**Figure 2:** Terraform Provider Registry

### Task 2: Install the Terraform AWS Provider

To install the Terraform AWS provider, and set the provider version in a way that is very similar to how you did for Terraform. To begin you need to let Terraform know to use the provider through a `required_providers` block in the `terraform.tf` file as seen below.

`terraform.tf`

```
terraform {  
  required_version = ">= 1.0.0"  
  required_providers {  
    aws = {  
      source  = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
    random = {  
      source  = "hashicorp/random"  
      version = "3.1.0"  
    }  
  }  
}
```





Note: You can always find the latest version of a provider on its > registry page at <https://registry.terraform.io>.

Run a `terraform init` to install the providers specified in the configuration

```
terraform init
```

### Task 3: View installed and required providers

If you ever would like to know which providers are installed in your working directory and those required by the configuration, you can issue a `terraform version` and `terraform providers` command.

```
terraform version
```

```
Terraform v1.0.10
on darwin_amd64
+ provider registry.terraform.io/hashicorp/aws v3.64.2
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

```
terraform providers
```

Providers required by configuration:

```
.
|-- provider[registry.terraform.io/hashicorp/aws] ~> 3.0
|-- provider[registry.terraform.io/hashicorp/random]
```

Providers required by state:

```
provider[registry.terraform.io/hashicorp/aws]
provider[registry.terraform.io/hashicorp/random]
```





## Lab: Terraform Provider Installation

Terraform relies on plugins called “providers” to interact with remote systems and expand functionality. Terraform configurations must declare which providers they require so that Terraform can install and use them. This is performed within a Terraform configuration block.

- Task 1: Install Terraform AWS Provider
- Task 2: Verify Terraform and AWS Provider version

### Task 1: Install Terraform AWS Provider

Terraform Providers are plugins that implement resource types for particular clouds, platforms and generally speaking any remote system with an API. Terraform configurations must declare which providers they require, so that Terraform can install and use them. Popular Terraform Providers include: AWS, Azure, Google Cloud, VMware, Kubernetes and Oracle.

In the next step we will install the Terraform AWS provider, and set the provider version in a way that is very similar to how you did for Terraform. To begin you need to let Terraform know to use the provider through a `required_providers` block in the `terraform.tf` file as seen below.

`terraform.tf`

```
terraform {  
  required_version = ">= 1.0.0"  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
  }  
}
```

Note: You can always find the latest version of a provider on its > registry page at <https://registry.terraform.io>.

Now that you have told Terraform to use this new provider you will have to run the `init` command. This will cause Terraform to notice the configuration change and download the provider.

```
terraform init
```

```
Initializing the backend...
```





```
Initializing provider plugins...
```

- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v3.62.0
- Using previously-installed hashicorp/random v3.1.0

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.
```

```
If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.
```

By default Terraform will always pull the latest provider if no version is set. However setting a version provides a way to ensure your Terraform code remains working in the event a newer version introduces a change that would not work with your existing code. To have more strict controls over the version you may want to require a specific version ( e.g. `required_version = "= 1.0.0"` ) or use the `~>`operator to only allow the right-most version number to increment.

## Task 2: Verify Terraform and AWS Provider version

To check the `terraform` version and provider version installed via `terraform init` run the `terraform version` command.

```
terraform version
```

```
Terraform v1.0.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```





## Lab: Terraform Resource Blocks

Terraform uses resource blocks to manage infrastructure, such as virtual networks, compute instances, or higher-level components such as DNS records. Resource blocks represent one or more infrastructure objects in your Terraform configuration. Most Terraform providers have a number of different resources that map to the appropriate APIs to manage that particular infrastructure type.

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Resource	AWS Provider	AWS Infrastructure
Resource 1	aws_instance	EC2 Instance
Resource 2	aws_security_group	Security Group
Resource 3	aws_s3_bucket	Amazon S3 bucket
Resource 4	aws_key_pair	EC2 Key Pair

When working with a specific provider, like AWS, Azure, or GCP, the resources are defined in the provider documentation. Each resource is fully documented in regards to the valid and required arguments required for each individual resource. For example, the `aws_key_pair` resource has a “Required” argument of `public_key` but optional arguments like `key_name` and `tags`. You’ll need to look at the provider documentation to understand what the supported resources are and how to define them in your Terraform configuration.

**Important** - Without `resource` blocks, Terraform is not going to create resources. All of the other block types, such as `variable`, `provider`, `terraform`, `output`, etc. are essentially supporting block types for the `resource` block.

- Task 1: View and understand an existing resource block in Terraform
- Task 2: Add a new resource to deploy an Amazon S3 bucket
- Task 3: Create a new AWS security group
- Task 4: Configure a resource from the random provider
- Task 5: Update the Amazon S3 bucket to use the random ID





## Task 1: View and understand an existing resource block in Terraform

Using the main.tf created in previous labs, look for a resource block that deploys a route table. The code should look something like this:

```
resource "aws_route_table" "public_route_table" {
    vpc_id = aws_vpc.vpc.id

    route {
        cidr_block      = "0.0.0.0/0"
        gateway_id      = aws_internet_gateway.internet_gateway.id
        #nat_gateway_id = aws_nat_gateway.nat_gateway.id
    }
    tags = {
        Name      = "demo_public_rtb"
        Terraform = "true"
    }
}
```

Let's look at the details in this resource block. First, all resource blocks will be declared using the resource block type. Next, you'll find the type of resource that is going to be deployed. In this case, it's `aws_route_table` which is part of the AWS provider. Finally, we gave this resource a local name of `public_route_table`.

Note: Your resource blocks must have a unique resource id (combination of resource type along with resource name). In our example, our resource id is `aws_route_table.public_route_table`, which is the combination of our resource type `aws_route_table` and resource name `public_route_table`. This naming and interpolation nomenclature is a powerful part of HCL that allows us to reference arguments from other resource blocks.

Within our resource block, we have arguments that are specific to the type of resource we are deploying. In our case, when we deploy a new AWS route table, there are certain arguments that are either `required` or `optional` that will apply to this resource. In our example, an AWS route table must be tied to a VPC, so we are providing the `vpc_id` argument and providing a value of our VPC using interpolation. Check out the `aws_route_table` documentation and see all the additional arguments that are available.

## Task 2: Add a new resource to deploy an Amazon S3 bucket

Ok, so now that we understand more about a resource block, let's create a new resource that will create an Amazon S3 bucket. In your main.tf file, add the following resource block:





```
resource "aws_s3_bucket" "my-new-S3-bucket" {
  bucket = "my-new-tf-test-bucket-bryan"

  tags = {
    Name      = "My S3 Bucket"
    Purpose   = "Intro to Resource Blocks Lab"
  }
}

resource "aws_s3_bucket_acl" "my_new_bucket_acl" {
  bucket = aws_s3_bucket.my-new-S3-bucket.id
  acl    = "private"
}
```

### Task 2.1.1

Run a `terraform plan` to see that this new Amazon S3 bucket will be added to our account. Don't worry, S3 buckets don't incur any fees unless you upload data to the bucket.

```
terraform plan
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

- + create

Terraform will perform the following actions:

```
# aws_s3_bucket.my-new-S3-bucket will be created
+ resource "aws_s3_bucket" "my-new-S3-bucket" {
    + acceleration_status      = (known after apply)
    + acl                      = "private"
    + arn                      = (known after apply)
    + bucket                   = "my-new-tf-test-bucket-bryan"
    + bucket_domain_name       = (known after apply)
    + bucketRegionalDomainName = (known after apply)
    + force_destroy            = false
    + hosted_zone_id          = (known after apply)
    + id                       = (known after apply)
    + region                   = (known after apply)
    + request_payer            = (known after apply)
    + tags                     = {
        + "Name"      = "My S3 Bucket"
        + "Purpose"   = "Intro to Resource Blocks Lab"
    }
    + tags_all                = {
```





```

    + "Name"      = "My S3 Bucket"
    + "Purpose"   = "Intro to Resource Blocks Lab"
}
+ website_domain           = (known after apply)
+ website_endpoint         = (known after apply)

+ versioning {
    + enabled     = (known after apply)
    + mfa_delete = (known after apply)
}
}

Plan: 1 to add, 0 to change, 0 to destroy.

```

Once the plan looks good, let's apply our configuration to the account by issuing a `terraform apply`

```
terraform apply
```

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
```

You will be prompted to confirm the changes before they're applied. Respond with `yes`.

### Task 2.2.1

Log into your AWS account and validate that the new Amazon S3 bucket exists.

### Task 3: Configure an AWS security group

Let's add one more resource block just to make sure you've got the hang of it. In this example, we're going to create a new Security Group that could be used for a web server to allow secure inbound connectivity over 443. If you are deploying EC2 instances or other resources in AWS, you'll probably need to define a custom security group.

In your main.tf file, add the following resource block:

```
resource "aws_security_group" "my-new-security-group" {
  name        = "web_server_inbound"
  description = "Allow inbound traffic on tcp/443"
  vpc_id      = aws_vpc.vpc.id
```





```

ingress {
  description = "Allow 443 from the Internet"
  from_port   = 443
  to_port     = 443
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

tags = {
  Name      = "web_server_inbound"
  Purpose   = "Intro to Resource Blocks Lab"
}

```

Notice that this resource, the `aws_security_group` requires completely different arguments. We're providing a name, a description, a VPC, and most importantly, the rules that we want to permit or restrict traffic. We also providing tags that will be added to the resource.

### Task 3.1.1

Run a `terraform plan` to see that this new Amazon S3 bucket will be added to our account. Don't worry, S3 buckets don't incur any fees unless you upload data to the bucket.

```
terraform plan
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
+ create

Terraform will perform the following actions:

```

# aws_security_group.my-new-security-group will be created
+ resource "aws_security_group" "my-new-security-group" {
    + arn                  = (known after apply)
    + description          = "Allow inbound traffic on tcp/443"
    + egress               = (known after apply)
    + id                   = (known after apply)
    + ingress              = [
        + {
            + cidr_blocks      = [
                + "0.0.0.0/0",
            ]
            + description       = "Allow 443 from the Internet"
            + from_port         = 443
            + ipv6_cidr_blocks = []
        }
    ]
}
```





```

        + prefix_list_ids  = []
        + protocol         = "tcp"
        + security_groups  = []
        + self              = false
        + to_port           = 443
    },
]
+ name                  = "web_server_inbound"
+ name_prefix          = (known after apply)
+ owner_id              = (known after apply)
+ revoke_rules_on_delete = false
+ tags                 = {
    + "Name"      = "web_server_inbound"
    + "Purpose"   = "Intro to Resource Blocks Lab"
}
+ tags_all              = {
    + "Name"      = "web_server_inbound"
    + "Purpose"   = "Intro to Resource Blocks Lab"
}
+ vpc_id                = "vpc-0407edc1d4962b00f"
}

Plan: 1 to add, 0 to change, 0 to destroy.

```

Once the plan looks good, let's apply our configuration to the account by issuing a `terraform apply`

```
terraform apply
```

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

You will be prompted to confirm the changes before they're applied. Respond with `yes`.

Log into AWS, browse to the EC2 or VPC console and verify the newly created Security Group.

#### **Task 4: Configure a resource from the random provider**

Ok, one more resource. But this time, it's not an AWS resource. Terraform supports many resources that don't interact with any other services. In fact, there is a provider that you can use just to create random data to be used in your Terraform.

Let's add a new resource block to Terraform using the `random` provider. In your `main.tf` file, add the





following resource block:

```
resource "random_id" "randomness" {  
    byte_length = 16  
}
```

### Task 4.1.1

Run a `terraform plan` to see that this new Amazon S3 bucket will be added to our account. Don't worry, S3 buckets don't incur any fees unless you upload data to the bucket.

```
terraform plan
```

You're probably going to get an output like this:

```
Error: Inconsistent dependency lock file  
  
The following dependency selections recorded in the lock file are  
inconsistent with the current configuration:  
- provider registry.terraform.io/hashicorp/random: required by this  
configuration but no version is selected  
  
To update the locked dependency selections to match a changed  
configuration, run:  
  terraform init -upgrade
```

Whoops....what happened? Well, we added a new resource that requires a provider that we haven't downloaded yet. Up until now, we've just been using the AWS provider, so adding AWS resources has worked fine. But now we need the hashicorp/random provider to use the `random_id` resource.

### Task 4.1.2

Let's run a `terraform init` so Terraform will download the provider we need. Notice in the output of the `terraform init` that you will see Terraform download the `hashicorp/random` provider.

```
terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Reusing previous version of `hashicorp/aws` from the dependency lock file
- Finding latest version of `hashicorp/random`...
- Using previously-installed `hashicorp/aws v3.63.0`





- Installing hashicorp/random v3.1.0...
- Installed hashicorp/random v3.1.0 (signed by HashiCorp)

Terraform has made some changes to the provider dependency selections recorded in the `.terraform.lock.hcl` file. Review those changes and commit them to your version control system **if** they represent changes you intended to make.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "`terraform plan`" to see any changes that are required **for** your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration **for** Terraform, rerun `this` command to reinitialize your working directory. If you forget, other commands will detect it and remind you to **do so if** necessary.

### Task 4.1.3

Run a `terraform plan` again to see that this new Amazon S3 bucket will be added to our account. Don't worry, S3 buckets don't incur any fees unless you upload data to the bucket.

```
terraform plan
```

Notice that Terraform is going to create a new resource for us now:

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
 + create

Terraform will perform the following actions:

```
# random_id.randomness will be created
+ resource "random_id" "randomness" {
    + b64_std      = (known after apply)
    + b64_url      = (known after apply)
    + byte_length  = 16
    + dec          = (known after apply)
    + hex          = (known after apply)
    + id           = (known after apply)
}
```





```
Plan: 1 to add, 0 to change, 0 to destroy.
```

### Task 4.1.4

Run a `terraform apply -auto-approve` to create the new resource. Remember that we're not actually creating anything on AWS, but generating a random 16 byte id.

```
terraform apply -auto-approve
```

You will see in the output that Terraform created a random id for us, and the value can be found in the output as shown below. Mine is `Htd2k6vC5Prb0xGeCBxAcQ` but yours will be different.

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create
```

Terraform will perform the following actions:

```
# random_id.randomness will be created
+ resource "random_id" "randomness" {
    + b64_std      = (known after apply)
    + b64_url      = (known after apply)
    + byte_length  = 16
    + dec          = (known after apply)
    + hex          = (known after apply)
    + id           = (known after apply)
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
random_id.randomness: Creating...
random_id.randomness: Creation complete after 0s [id=
Htd2k6vC5Prb0xGeCBxAcQ]
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

### Task 5: Update the Amazon S3 bucket to use the random ID

Now that we have a `random_id` being created by Terraform, let's see how we can use that for other resources. If you're not aware, Amazon S3 bucket names have to be globally unique, meaning nobody in the world can create a bucket with the same name as an existing bucket. That means if I create a bucket named “my-cool-s2-bucket”, nobody else can create a bucket with the same name. This is





## Hands-On Labs

where the `random_id` might come in handy, so let's update the name of our bucket to use a random ID.

In your `main.tf` file, find the resource block where you created a new Amazon S3 bucket. It's the code we added in Task 2 above. Modify the `bucket` argument to include the following:

```
resource "aws_s3_bucket" "my-new-S3-bucket" {
  bucket = "my-new-tf-test-bucket-${random_id.randomness.hex}"

  tags = {
    Name      = "My S3 Bucket"
    Purpose   = "Intro to Resource Blocks Lab"
  }
}
```

### Task 5.1.1

Run a `terraform plan` again to see that Terraform is going to replace our Amazon S3 bucket in our account because the name is changing. The name will now end with our random id that we created using the `random_id` resource.

```
terraform plan
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
-/+ destroy and then create replacement

Terraform will perform the following actions:

```
# aws_s3_bucket.my-new-S3-bucket must be replaced
-/+ resource "aws_s3_bucket" "my-new-S3-bucket" {
  + acceleration_status          = (known after apply)
  ~ arn                          = "arn:aws:s3:::my-new-tf-test-bucket-
    bryan" -> (known after apply)
  ~ bucket                        = "my-new-tf-test-bucket-bryan" -> "my
    -new-tf-test-bucket-Htd2k6vC5Prb0xGeCBxAcQ" # forces replacement
  ~ bucket_domain_name           = "my-new-tf-test-bucket-bryan.s3.
    amazonaws.com" -> (known after apply)
  ~ bucketRegionalDomainName     = "my-new-tf-test-bucket-bryan.s3.
    amazonaws.com" -> (known after apply)
  ~ hostedZoneId                 = "Z3AQBSTGFYJSTF" -> (known after
    apply)
  ~ id                            = "my-new-tf-test-bucket-bryan" -> (
    known after apply)
  ~ region                        = "us-east-1" -> (known after apply)
```





```

~ request_payer           = "BucketOwner" -> (known after apply)
~ tags                     = {
  + "Random"   = "Htd2k6vC5Prb0xGeCBxAcQ"
    # (2 unchanged elements hidden)
}
~ tags_all                = {
  + "Random"   = "Htd2k6vC5Prb0xGeCBxAcQ"
    # (2 unchanged elements hidden)
}
+ website_domain          = (known after apply)
+ website_endpoint         = (known after apply)
  # (2 unchanged attributes hidden)

~ versioning {
  ~ enabled    = false -> (known after apply)
  ~ mfa_delete = false -> (known after apply)
}
}

Plan: 1 to add, 0 to change, 1 to destroy.

```

### Task 5.1.2

Now that we're done with going over the Resource block, feel free to delete the resources that we created in this lab. These resources include:

- Amazon S3 bucket
- Security Group
- Random ID

Once you deleted them from your Terraform configuration file, go ahead and run a `terraform plan` and `terraform apply` to delete the resources from your account.





## Lab: Introduction to the Terraform Variables Block

As you begin to write Terraform templates with a focus on reusability and DRY development (don't repeat yourself), you'll quickly begin to realize that variables are going to simplify and increase usability for your Terraform configuration. Input variables allow aspects of a module or configuration to be customized without altering the module's own source code. This allows modules to be shared between different configurations.

Input variables (commonly referenced as just 'variables') are often declared in a separate file called `variables.tf`, although this is not required. Most people will consolidate variable declaration in this file for organization and simplification of management. Each variable used in a Terraform configuration must be declared before it can be used. Variables are declared in a variable block - one block for each variable. The variable block contains the variable name, most importantly, and then often includes additional information such as the type, a description, a default value, and other options.

The variable block follows the following pattern:

### Template

```
variable "<VARIABLE_NAME>" {
  # Block body
  type = <VARIABLE_TYPE>
  description = <DESCRIPTION>
  default = <EXPRESSION>
  sensitive = <BOOLEAN>
  validation = <RULES>
}
```

### Example

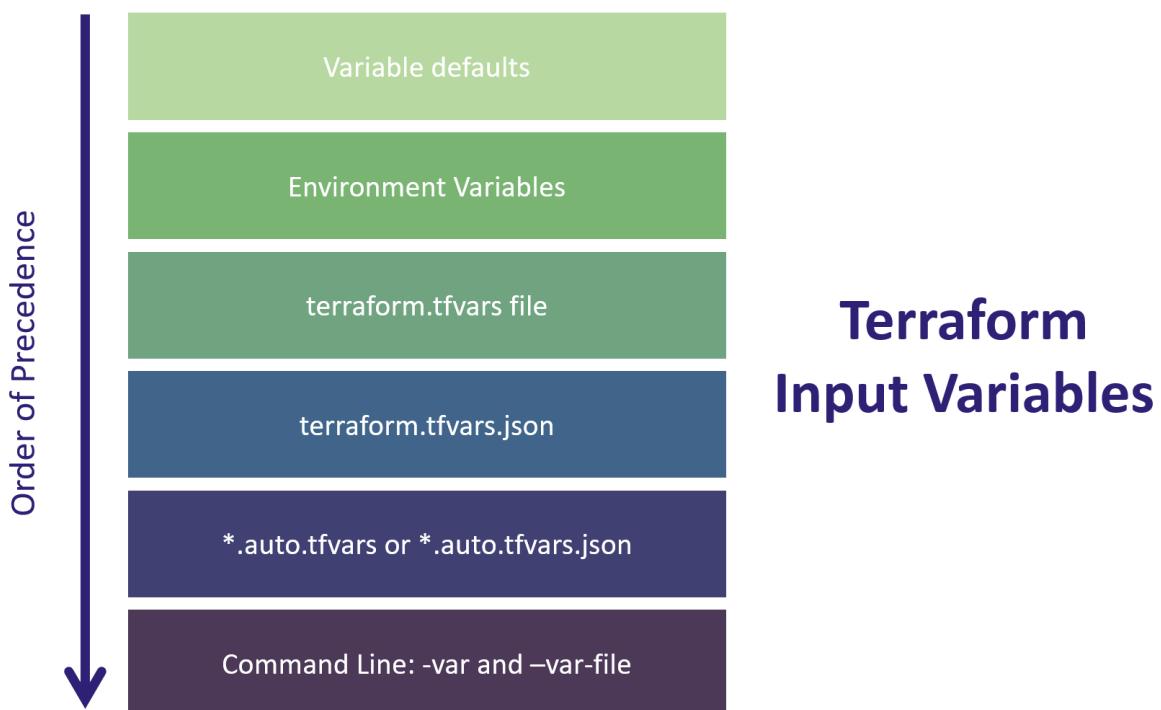
```
variable "aws_region" {
  type      = string
  description = "region used to deploy workloads"
  default   = "us-east-1"
  validation {
    condition    = can(regex("^us-", var.aws_region))
    error_message = "The aws_region value must be a valid region in the USA, starting with \"us-\"."
  }
}
```





{}

The value of a Terraform variable can be set multiple ways, including setting a default value, interactively passing a value when executing a terraform plan and apply, using an environment variable, or setting the value in a `.tfvars` file. Each of these different options follows a strict order of precedence that Terraform uses to set the value of a variable.



**Figure 1:** Terraform Input Variables - Order of Precedence

- Task 1: Add a new VPC resource block with static values
- Task 2: Define new variable blocks to declare new variables
- Task 3: Modify the value of the variable by adding defaults

### Task 1: Add a new VPC resource block with static values

Before we can add any meaningful new variables, we should add a new resource to use as a demo. Let's add a new VPC resource that we'll use in this lab. In the `main.tf` file, add the following code at the bottom. ***Do not remove the existing VPC resource***





```
resource "aws_subnet" "variables-subnet" {
    vpc_id              = aws_vpc.vpc.id
    cidr_block          = "10.0.250.0/24"
    availability_zone   = "us-east-1a"
    map_public_ip_on_launch = true

    tags = {
        Name      = "sub-variables-us-east-1a"
        Terraform = "true"
    }
}
```

### Task 1.1

Run `terraform plan` to validate the changes to your configuration. You should see that Terraform is going to add a new VPC resource that we just added.

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

- + create

Terraform will perform the following actions:

```
# aws_subnet.variables-subnet will be created
+ resource "aws_subnet" "variables-subnet" {
    + arn                  = (known after apply)
    + assign_ipv6_address_on_creation = false
    + availability_zone       = "us-east-1a"
    + availability_zone_id    = (known after apply)
    + cidr_block              = "10.0.250.0/24"
    + id                     = (known after apply)
    + ipv6_cidr_block_association_id = (known after apply)
    + map_public_ip_on_launch = true
    + owner_id                = (known after apply)
    + tags
        + "Name"      = "sub-variables-us-east-1a"
        + "Terraform" = "true"
    }
    + tags_all               = {
        + "Name"      = "sub-variables-us-east-1a"
        + "Terraform" = "true"
    }
    + vpc_id                 = "vpc-069bc66bf87daccd6"
}
```

Plan: 1 to add, 0 to change, 0 to destroy.





Once the plan looks good, let's apply our configuration by issuing a `terraform apply`

```
$ terraform apply

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
```

You will be prompted to confirm the changes before they're applied. Respond with `yes`.

## Task 2: Define new variable blocks to declare new variables

Now that we have a new resources to use, let's change some of those static values to use variables instead. In your `variables.tf` file, add the following variable blocks anywhere in the file:

```
variable "variables_sub_cidr" {
  description = "CIDR Block for the Variables Subnet"
  type        = string
}

variable "variables_sub_az" {
  description = "Availability Zone used Variables Subnet"
  type        = string
}

variable "variables_sub_auto_ip" {
  description = "Set Automatic IP Assignment for Variables Subnet"
  type        = bool
}
```

### Task 2.1

In your `main.tf` file, let's remove our static values and replace them with our new variables to make our configuration a bit more flexible. Update the configuration file so the new VPC resource reflects the changes shown below:

```
resource "aws_subnet" "variables-subnet" {
  vpc_id           = aws_vpc.vpc.id
  cidr_block       = var.variables_sub_cidr
  availability_zone = var.variables_sub_az
```





```
map_public_ip_on_launch = var.variables_sub_auto_ip

tags = {
  Name      = "sub-variables-${var.variables_sub_az}"
  Terraform = "true"
}
```

## Task 2.2

We now have new variables declared in our `variables.tf` file along with a modified resource (our new subnet) to use those variables. Let's run a `terraform plan` to see what happens with our new resource and variables. Did you get something similar to what is shown below?

```
$ terraform plan
var.variables_sub_auto_ip
  Set Automatic IP Assignment for variables Subnet

Enter a value:
```

At this point, we've declared the variables and have referenced them in our subnet resource, but Terraform has no idea what value you want to use for the new variables. Let's provide it with our values.

- for `var.variables_sub_auto_ip`, type in “true” and press enter
- for `var.variables_sub_az`, type in “us-east-1a” and press enter
- for `var.variables_sub_cidr`, type in “10.0.250.0/24” and press enter

Now that Terraform knows the values we want to use, it can proceed with our configuration. When the `terraform plan` is completed, it should have found that there are no changes needed to our infrastructure because we've defined the same values for our subnet, but using variables instead.

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

## Task 3: Modify the value of the variable by adding defaults

Another common method of providing values for variables is by setting default values in the variable block. If no other values are provided during a `terraform plan` or `terraform apply`, then Ter-





Terraform will use the default values. This is especially handy when you have variables where the value doesn't often change.

In the `variables.tf` file, modify the new variables we added earlier to now include a default argument and value.

```
variable "variables_sub_cidr" {
  description = "CIDR Block for the Variables Subnet"
  type        = string
  default     = "10.0.202.0/24"
}

variable "variables_sub_az" {
  description = "Availability Zone used for Variables Subnet"
  type        = string
  default     = "us-east-1a"
}

variable "variables_sub_auto_ip" {
  description = "Set Automatic IP Assignment for Variables Subnet"
  type        = bool
  default     = true
}
```

### Task 3.1

Let's test our new defaults by running another `terraform plan`. You should see that Terraform no longer asks us for a value for the variables since it will just use the default value if you don't specify one using another method. And since we provided different values, Terraform wants to update our infrastructure.

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
`-/+` destroy and then create replacement

Terraform will perform the following actions:

```
# aws_subnet.variables-subnet must be replaced
-/+ resource "aws_subnet" "variables-subnet" {
    ~ arn                      = "arn:aws:ec2:us-east-1:603991114860:subnet/subnet-0b424eed2dc2822d0" -> (known after apply)
    ~ availability_zone_id      = "use1-az6" -> (known after apply)
```





## Hands-On Labs

```
~ cidr_block           = "10.0.250.0/24" -> "
  10.0.202.0/24" # forces replacement
~ id                   = "subnet-0b424eed2dc2822d0" -> (
  known after apply)
+ ipv6_cidr_block_association_id = (known after apply)
- map_customer_owned_ip_on_launch = false -> null
~ owner_id             = "603991114860" -> (known after
  apply)
tags                  = {
  "Name"      = "sub-variables-us-east-1a"
  "Terraform" = "true"
}
# (5 unchanged attributes hidden)
}
```

Plan: 1 to add, 0 to change, 1 to destroy.

### Task 3.1

Let's go ahead and apply our new configuration, which will replace the subnet with one using the CIDR block of “10.0.202.0/24”. Run a `terraform apply`. Don't forget to accept the changes by typing `yes`.





## Lab: Introduction to the Terraform Locals Block

Locals blocks (often referred to as locals) are defined values in Terraform that are used to reduce repetitive references to expressions or values. Locals are very similar to traditional input variables and can be referred to throughout your Terraform configuration. Locals are often used to give a name to the result of an expression to simplify your code and make it easier to read.

Locals are not set directly by the user/machine executing the Terraform configuration, and the values don't change between or during the Terraform workflow ([init](#), [plan](#), [apply](#)).

Locals are defined in a `locals` block (plural) and include named local variables with their defined values. Each locals block can contain one or more local variables. Locals are then referenced in your configuration using interpolation using `local.<name>` (note `local` and not `locals`). The syntax of a locals block is as follows:

### Template

```
locals {  
  # Block body  
  local_variable_name = <EXPRESSION OR VALUE>  
  local_variable_name = <EXPRESSION OR VALUE>  
}
```

### Example

```
locals {  
  time = timestamp()  
  application = "api_server"  
  server_name = "${var.account}-${local.application}"
```

- Task 1: Define the Name of an EC2 Instance using a Local Variable

#### Task 1: Define the Name of an EC2 Instance using a Local Variable

Before we can use a locals variable, it needs to be defined in our configuration file. Locals are often defined right in the configuration file where they will be used. In the `main.tf` file, add the following locals block to the configuration file:





```
locals {
  team = "api_mgmt_dev"
  application = "corp_api"
  server_name = "ec2-${var.environment}-api-${var.variables_sub_az}"
}
```

Next, let's update the configuration of our web server resource to use the locals variable to define the name. Modify the `web_server` resource in `main.tf` so it matches the following:

```
resource "aws_instance" "web_server" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnets["public_subnet_1"].id
  tags = [
    { Name = local.server_name }
    { Owner = local.team }
    { App = local.application }
  ]
}
```

### Task 1.1

Run a `terraform plan` to see the changes to our web server. You should see that the EC2 instance will not be destroyed, but the name of the server (tag) will be updated to reflect the value of our local variables.

```
terraform plan
```

```
<add output here>
```

### Task 1.2

Let's go ahead and commit our code to validate Terraform makes the desired changes to the AWS infrastructure. Run a `terraform apply` to apply the changes. Log into AWS and validate the changes were applied.

Once validated, feel free to remove the `locals` block and tags on the EC2 instance and then apply the configuration to revert back to the original values. However, you can leave everything in your Terraform configuration if you'd like.





## Lab: Introduction to the Terraform Data Block

Data sources are used in Terraform to load or query data from APIs or other Terraform workspaces. You can use this data to make your project's configuration more flexible, and to connect workspaces that manage different parts of your infrastructure. You can also use data sources to connect and share data between workspaces in Terraform Cloud and Terraform Enterprise.

To use a data source, you declare it using a [data](#) block in your Terraform configuration. Terraform will perform the query and store the returned data. You can then use that data throughout your Terraform configuration file where it makes sense.

Data Blocks within Terraform HCL are comprised of the following components:

- Data Block - “resource” is a top-level keyword like “for” and “while” in other programming languages.
- Data Type - The next value is the type of the resource. Resources types are always prefixed with their provider (aws in this case). There can be multiple resources of the same type in a Terraform configuration.
- Data Local Name - The next value is the name of the resource. The resource type and name together form the resource identifier, or ID. In this lab, one of the resource IDs is `aws_instance.web`. The resource ID must be unique for a given configuration, even if multiple files are used.
- Data Arguments - Most of the arguments within the body of a resource block are specific to the selected resource type. The resource type’s documentation lists which arguments are available and how their values should be formatted.

Example: A data block requests that Terraform read from a given data source (“aws\_ami”) and export the result under the given local name (“example”). The name is used to refer to this resource from elsewhere in the same Terraform module.

## Template

```
data "<DATA TYPE>" "<DATA LOCAL NAME>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

- Task 1: Add a new data source to query the current AWS region being used
- Task 2: Update the Terraform configuration file to use the new data source





- Task 3: View the data source used to retrieve the availability zones within the region
- Task 4: Validate the data source is being used in the Terraform configuration file
- Task 5: Create a new data source for querying a different Ubuntu image
- Task 6: Make the aws\_instance web\_server use the Ubuntu image returned by the data source

### **Task 1: Add a new data source to query the current AWS region being used**

Modify the main.tf file to include a new data source block. This simple data source will use the AWS credentials and determine the AWS region that you are using.

```
#Retrieve the AWS region
data "aws_region" "current" { }
```

### **Task 2: Update the Terraform configuration file to use the new data source**

Now that we know Terraform will query AWS for the current region, we can now use that information within our configuration file. In our main.tf file, you'll find the resource block where we are creating the VPC. Within that block, you'll see that we are adding tags to the VPC for easily identification. Let's add a "Region" tag and use the results of our data source we just added.

Modify the VPC resource block to add another tag and get the value from our data source. Terraform will query AWS, grab the current region, and add the value to our new tag.

```
#Define the VPC
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name          = var.vpc_name
    Environment   = "demo_environment"
    Terraform     = "true"
    Region        = data.aws_region.current.name
  }
}
```

Notice the formatting of our reference to the data block. We'll go into more detail when we learn about Variables, but the syntax to refer to a data block is as follows:

`data.<type>.<name>.<attribute>`





In our example, `data` is used since we're referring to a data block. The “**type**” is `aws_region`, the “**name**” is `current`, and the “**attribute**” is `name`. You can simply look at the data block and see how this is all put together. Finally, `name` is one of the attributes that is exported by the `aws_region` data source. Make sure to check out the official documentation for a data source to learn about the different attributes available for the one you’re using.

### Task 3: View the data source used retrieve the availability zones within the region

If you didn’t noticed already, we were already using a data source to retrieve the availability zones in the region we are using. Towards the top of the main.tf file, look for the following code:

```
#Retrieve the list of AZs in the current AWS region
data "aws_availability_zones" "available" {}
```

It looks very similar to our first task, but this is retrieving different data for us. Obtaining this data might be helpful if you are building a fleet of web servers and you want to ensure each one is deployed in a different AZ. The benefit of using a data source, as opposed to manually typing in the availability zones, is that you can run the exact same Terraform in multiple regions and it would work without modifications, since the AZs are dynamically obtained by the data source.

### Task 4: Validate the data source is being used in the Terraform configuration filee

Let’s take a look at how we’re already using this data source in our deployment. Look in the main.tf and find the resource block that is creating our private subnets. When you create a subnet, one of the required parameters is choosing an availability zone where the subnet will be created. Rather than hard code this, we can use our data source to dynamically choose an availability zone.

```
#Deploy the private subnets
resource "aws_subnet" "private_subnets" {
  for_each           = var.private_subnets
  vpc_id             = aws_vpc.vpc.id
  cidr_block         = cidrsubnet(var.vpc_cidr, 8, each.value)
  availability_zone = tolist(data.aws_availability_zones.available.names)[
    each.value]

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}
```





## Task 5: Create a new data source for querying a different Ubuntu image

### Step 5.1.1

Add an aws\_ami data source called “ubuntu\_16\_04” in `main.tf`. It will find the most recent instance of a Ubuntu 16.04 AMI from Canonical (owner ID 099720109477).

```
# Terraform Data Block - Lookup Ubuntu 16.04
data "aws_ami" "ubuntu_16_04" {
    most_recent = true

    filter {
        name    = "name"
        values  = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
    }

    owners = ["099720109477"]
}
```

## Task 6: Modify the aws\_instance so it uses the returned AMI

Edit the aws\_instance in `main.tf` so that its ami argument uses the AMI returned by the data source.

```
resource "aws_instance" "web_server" {
    ami                      = data.aws_ami.ubuntu_16_04.id
    instance_type            = "t2.micro"
    subnet_id                = aws_subnet.public_subnets["public_subnet_1"]
    ".id"
    security_groups          = [aws_security_group.vpc-ping.id]
    associate_public_ip_address = true
    tags = {
        Name = "Web EC2 Server"
    }
}
```

### Step 6.1.1

Run `terraform apply` one last time to apply the changes you made.

```
terraform apply
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
```





```
Enter a value: yes
```

```
aws_instance.web_server: Destroying... [id=i-007601704642290cc]
aws_instance.web_server: Still destroying... [id=i-007601704642290cc, 10s
  elapsed]
aws_instance.web_server: Still destroying... [id=i-007601704642290cc, 20s
  elapsed]
aws_instance.web_server: Still destroying... [id=i-007601704642290cc, 30s
  elapsed]
aws_instance.web_server: Destruction complete after 40s
aws_instance.web_server: Creating...
aws_instance.web_server: Still creating... [10s elapsed]
aws_instance.web_server: Still creating... [20s elapsed]
aws_instance.web_server: Still creating... [30s elapsed]
aws_instance.web_server: Creation complete after 31s [id=i-0566
  fade24f7cd155]
```

```
Apply complete!! Resources: 1 added, 0 changed, 1 destroyed.
```





## Lab: Terraform Configuration Block

Terraform relies on plugins called “providers” to interact with remote systems and expand functionality. Terraform configurations must declare which providers they require so that Terraform can install and use them. This is performed within a Terraform configuration block.

- Task 1: Check Terraform version
- Task 2: Require specific versions of Terraform

Template:

```
terraform {  
  # Block body  
  <ARGUMENT> = <VALUE>  
}
```

Example:

```
terraform {  
  required_version = ">= 1.0.0"  
}
```

### Task 1: Check Terraform version

Run the following command to check the Terraform version:

```
terraform -version
```

You should see:

```
Terraform v1.0.8
```

### Task 2: Require specific versions of Terraform

Create a file titled `terraform.tf` to define a minimum version of Terraform to be used.

```
terraform.tf
```

```
terraform {  
  required_version = ">= 1.0.0"  
}
```





```
terraform init
```

```
Terraform v1.0.8
```

This informs Terraform that it must be at least of version 1.0.0 to run the code. If Terraform is an earlier version it will throw an error.

Update the `required_version` line to "`=1.0.0`" and run `terraform init`.

What happened when you changed the version requirement?

```
Error: Unsupported Terraform Core version
```

```
on terraform.tf line 2, in terraform:
 2:   required_version = "= 1.0.0"
```

```
This configuration does not support Terraform version 1.0.8. To proceed,
either choose another supported Terraform
version or update this version constraint. Version constraints are
normally set for good reason, so updating the
constraint may lead to other errors or unexpected behavior.
```

At this point you have now stated that Terraform can only run this code if its own version is equal to 1.0.0.

Note: Make sure the required version is set back to "`>=1.0.0`" and you have run `terraform init` again before proceeding.

## Reference

Terraform Settings

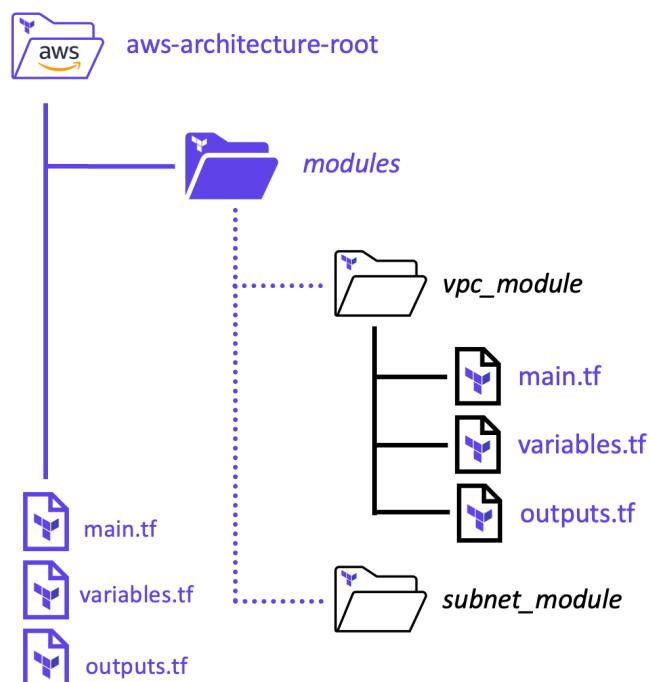




## Lab: Introduction to the Module Block

A module is used to combine resources that are frequently used together into a reusable container. Individual modules can be used to construct a holistic solution required to deploy applications. The goal is to develop modules that can be reused in a variety of different ways, therefore reducing the amount of code that needs to be developed. Modules are called by a [parent](#) or [root](#) module, and any modules called by the parent module are known as [child](#) modules.

Modules can be sourced from a number of different locations, including remote, such as the Terraform module registry, or locally within a folder. While not required, local modules are commonly saved in a folder named `modules`, and each module is named for its respective function inside that folder. An example of this can be found in the diagram below:



**Figure 1:** Module Structure

Modules are defined in a `module` block with a unique name for each module. Within the module block, the `source` indicates the local path of the module or the remote source where Terraform should download the module. You can also specify the `version` of the module to use, along with inputs that are passed to the child module.





## Template

```
module "<MODULE_NAME>" {
  # Block body
  source = <MODULE_SOURCE>
  <INPUT_NAME> = <DESCRIPTION> #Inputs
  <INPUT_NAME> = <DESCRIPTION> #Inputs
}
```

## Example

```
module "website_s3_bucket" {
  source = "./modules/aws-s3-static-website-bucket"

  bucket_name = var.s3_bucket_name
  aws_region = "us-east-1"

  tags = {
    Terraform = "true"
    Environment = "certification"
  }
}
```

- Task 1: Create a new module block to call a remote module

In order to call a module, let's create a new module block. In the `main.tf` file, add the following code:

```
module "subnet_addrs" {
  source  = "hashicorp/subnets/cidr"
  version = "1.0.0"

  base_cidr_block = "10.0.0.0/22"
  networks = [
    {
      name      = "module_network_a"
      new_bits = 2
    },
    {
      name      = "module_network_b"
      new_bits = 2
    },
  ]
}
```





```
output "subnet_addrs" {  
    value = module.subnet_addrs.network_cidr_blocks  
}
```

## Task 1.1

Now that we've added the new module block, let's first run a `terraform init` so Terraform can download the referenced module for us.

```
$ terraform init  
Initializing modules...  
Downloading hashicorp/subnets/cidr 1.0.0 for subnet_addrs...  
- subnet_addrs in .terraform/modules/subnet_addrs  
  
Initializing the backend...  
  
Initializing provider plugins...  
  
Terraform has been successfully initialized!  
  
You may now begin working with Terraform. Try running "terraform plan" to  
see  
any changes that are required for your infrastructure. All Terraform  
commands  
should now work.  
  
If you ever set or change modules or backend configuration for Terraform,  
rerun this command to reinitialize your working directory. If you forget,  
other  
commands will detect it and remind you to do so if necessary.
```

## Task 1.2

Now that the module has been downloaded, let's apply our new configuration. Keep in mind that this module only calculates subnets for us and returns those subnets as an output. It doesn't create any resources in AWS. Run a `terraform apply -auto-approve` to apply the new configuration.

```
$ terraform apply -auto-approve  
  
No changes. Your infrastructure matches the configuration.
```





```
Terraform has compared your real infrastructure against your configuration  
and found no differences, so no changes are needed.
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Outputs:

```
subnet_addrs = tomap({  
  "module_network_a" = "10.0.0.0/24"  
  "module_network_b" = "10.0.1.0/24"  
)
```





## Lab: Introduction to the Terraform Output Block

Terraform output values allow you to export structured data about your resources. You can use this data to configure other parts of your infrastructure with automation tools, or as a data source for another Terraform workspace. Outputs are also necessary to share data from a child module to your root module.

As with all other blocks in HashiCorp Configuration Language (HCL), the output block has a particular syntax that needs to be followed when creating output blocks. Each output name should be unique. The syntax looks like this:

### Template

```
output "<NAME>" {  
    # Block body  
    value = <EXPRESSION> # Argument  
}
```

### Example

```
output "web_server_ip" {  
    description = "Public IP Address of Web Server on EC2"  
    value        = aws_instance.web_server.public_ip  
    sensitive   = true  
}
```

- Task 1: Add Output Block to Export Attributes of Resources
- Task 2: Output Meaningful Data using Static and Dynamic Values
- Task 3: Generate Machine-Readable Outputs in JSON

### Task 1: Add Output Block to Export Attributes of Resources

In the same working directory that contains our `main.tf` and `variables.tf` files, create a new file called `outputs.tf`. This file is commonly used to store all output blocks within your working directory. In the new file, add the following code:





```
output "hello-world" {  
  description = "Print a Hello World text output"  
  value = "Hello World"  
}  
  
output "vpc_id" {  
  description = "Output the ID for the primary VPC"  
  value = aws_vpc.vpc.id  
}
```

### Task 1.1

After saving the new `outputs.tf` file, run a `terraform apply -auto-approve` to see our new outputs for our infrastructure.

```
No changes. Your infrastructure matches the configuration.  
  
Terraform has compared your real infrastructure against your configuration  
and found no  
differences, so no changes are needed.  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
hello-world = "Hello World"  
vpc_id = "vpc-058d23c9d5d2f70b5"
```

## Task 2: Output Meaningful Data using Static and Dynamic Values

Resource attribute data can also be included within a string to include additional information or create a particular output, such as a website URL or database connection string. Modify the `outputs.tf` file to include the additional output blocks as shown below:

```
output "public_url" {  
  description = "Public URL for our Web Server"  
  value = "https://${aws_instance.web_server.private_ip}:8080/index.html"  
}  
  
output "vpc_information" {  
  description = "VPC Information about Environment"
```





```
    value = "Your ${aws_vpc.vpc.tags.Environment} VPC has an ID of ${aws_vpc
        .vpc.id}"
}
```

## Task 2.1

After saving the new `outputs.tf` file, run a `terraform apply -auto-approve` to see our new output. The output now provides a full URL that contains the IP address of our public web server. We also have an additional string that provides information about our VPC as well.

```
No changes. Your infrastructure matches the configuration.
```

```
Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes
are needed.
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
hello-world = "Hello World"
public_url = "https://10.0.101.10:8080/index.html"
vpc_id = "vpc-058d23c9d5d2f70b5"
vpc_information = "Your demo_environment VPC has an ID of vpc-058
d23c9d5d2f70b5"
```

## Task 2.2

If you ever need to read the values of your Terraform outputs but might be nervous about running a `terraform apply`, you can use the `terraform output` command to view them.

Run a `terraform output` and view the list of outputs from your configuration:

```
$ terraform output
hello-world = "Hello World"
public_url = "https://10.0.101.10:8080/index.html"
vpc_id = "vpc-058d23c9d5d2f70b5"
vpc_information = "Your demo_environment VPC has an ID of vpc-058
d23c9d5d2f70b5"
```





### Task 3: Generate Machine-Readable Outputs in JSON

When working with Terraform in your organization, it is common practice to use an automation tool to automate your terraform deployments. But don't fret! We can still use outputs and make them "machine-readable" so other automation tools can parse the information, if needed.

Run a `terraform output -json` and view the list of outputs in JSON format from your configuration:

```
terraform output -json
{
  "hello-world": {
    "sensitive": false,
    "type": "string",
    "value": "Hello World"
  },
  "public_url": {
    "sensitive": false,
    "type": "string",
    "value": "https://10.0.101.10:8080/index.html"
  },
  "vpc_id": {
    "sensitive": false,
    "type": "string",
    "value": "vpc-058d23c9d5d2f70b5"
  },
  "vpc_information": {
    "sensitive": false,
    "type": "string",
    "value": "Your demo_environment VPC has an ID of vpc-058d23c9d5d2f70b5"
  }
}
```





## Lab: Commenting Terraform Code

To make our code easier to understand for others who might want to contribute we may want to add a comment to explain what a resource or a particular code block is doing. The Terraform language supports three different syntaxes for comments:

```
# begins a single-line comment, ending at the end of the line.
```

```
// also begins a single-line comment, as an alternative to #.
```

```
/* and */ are start and end delimiters for a comment that might span over  
multiple lines.
```

- Task 1: Add Single Line Comment to Terraform Configuration
- Task 2: Add Multi-Line Comment to Terraform Configuration

### Task 1: Add Single Line Comment to Terraform Configuration

Update your `main.tf` to include a single line comment at the top of your file.

```
# IaC Buildout for Terraform Associate Exam  
  
# Configure the AWS Provider  
provider "aws" {  
  region = "us-east-1"  
}
```

Run a `terraform plan` to validate that adding a single line comment has no effect on the configuration and thus will return a `No Change` plan.

```
terraform plan
```

```
No changes. Infrastructure is up-to-date.
```

Note: There is no harm in running a `terraform plan` after adding any items into a Terraform configuration file. In fact it can be quite useful to run to validate that your configuration changes will not have any impact on your deployments.

### Task 2: Add Multi-Line Comment to Terraform Configuration

Update your `main.tf` to include a single line comment at the top of your file.





```
/*
Name: IaC Buildout for Terraform Associate Exam
Description: AWS Infrastructure Buildout
Contributors: Bryan and Gabe
*/
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}
```

Run a `terraform plan` to validate that adding a single line comment has no effect on the configuration and thus will return a `No Change` plan.

```
terraform plan
```

```
No changes. Infrastructure is up-to-date.
```

The `#` single-line comment style is the default comment style and should be used in most cases. Automatic configuration formatting tools may automatically transform `//` comments into `#` comments, since the double-slash style is not idiomatic.

## Reference

Terraform HCL Comments





## Lab: Terraform Provider Installation

Terraform relies on plugins called “providers” to interact with remote systems and expand functionality. Terraform configurations must declare which providers they require so that Terraform can install and use them. This is performed within a Terraform configuration block.

- Task 1: Install Terraform AWS Provider
- Task 2: Verify Terraform and AWS Provider version

### Task 1: Install Terraform AWS Provider

Terraform Providers are plugins that implement resource types for particular clouds, platforms and generally speaking any remote system with an API. Terraform configurations must declare which providers they require, so that Terraform can install and use them. Popular Terraform Providers include: AWS, Azure, Google Cloud, VMware, Kubernetes and Oracle.

In the next step we will install the Terraform AWS provider, and set the provider version in a way that is very similar to how you did for Terraform. To begin you need to let Terraform know to use the provider through a `required_providers` block in the `terraform.tf` file as seen below.

`terraform.tf`

```
terraform {  
  required_version = ">= 1.0.0"  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
  }  
}
```

Note: You can always find the latest version of a provider on its > registry page at <https://registry.terraform.io>.

Now that you have told Terraform to use this new provider you will have to run the `init` command. This will cause Terraform to notice the configuration change and download the provider.

```
terraform init
```

```
Initializing the backend...
```





```
Initializing provider plugins...
```

- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v3.62.0
- Using previously-installed hashicorp/random v3.1.0

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.
```

```
If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.
```

By default Terraform will always pull the latest provider if no version is set. However setting a version provides a way to ensure your Terraform code remains working in the event a newer version introduces a change that would not work with your existing code. To have more strict controls over the version you may want to require a specific version ( e.g. `required_version = "= 1.0.0"` ) or use the `~>`operator to only allow the right-most version number to increment.

## Task 2: Verify Terraform and AWS Provider version

To check the `terraform` version and provider version installed via `terraform init` run the `terraform version` command.

```
terraform version
```

```
Terraform v1.0.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```





## Lab: Multiple Terraform Providers

Due to the plug-in based architecture of Terraform providers, it is easy to install and utilize multiple providers within the same Terraform configuration. Along with the already configured AWS provider, we will install both the HTTP and Random providers.

- Task 1: Install Terraform HTTP provider version
- Task 2: Configure Terraform HTTP Provider
- Task 3: Install Terraform Random provider version
- Task 4: Configure Terraform Random Provider
- Task 5: Install Terraform Local provider version

### Task 1: Install Terraform HTTP provider version

The HTTP provider is a utility provider for interacting with generic HTTP servers as part of a Terraform configuration.

As with any provider in can be installed by specifying it with with a terraform configuration block.

A screenshot of the HashiCorp Registry website. The top navigation bar includes links for 'Browse', 'Publish', and 'Sign-in'. Below the header, a search bar says 'Search Providers and Modules'. The main content area shows the 'http' provider page under the 'hashicorp' organization. The left sidebar has a 'HTTP DOCUMENTATION' section with a 'Filter' input field and a link to the 'http provider'. The main content area has a title 'HTTP Provider' and a description: 'The HTTP provider is a utility provider for interacting with generic HTTP servers as part of a Terraform configuration.' It also states: 'This provider requires no configuration. For information on the resources it provides, see the navigation bar.' At the bottom right, there is a 'How to use this provider' section with code examples for Terraform 0.13+ and later versions. The code shows how to declare the provider in a Terraform configuration file.

```
terraform {  
    required_providers {  
        http = {  
            source = "hashicorp/http"  
            version = "2.1.0"  
        }  
    }  
  
    provider "http" {  
        # Configuration options  
    }  
}
```

**Figure 1:** HTTP Provider

Add the following to the `terraform.tf` file





```
terraform {  
  required_providers {  
    http = {  
      source = "hashicorp/http"  
      version = "2.1.0"  
    }  
  }  
}
```

To install the provider execute a `terraform init`

```
terraform init
```

Initializing the backend...

Initializing provider plugins...

- Reusing previous version of hashicorp/aws from the dependency lock file
- Finding hashicorp/http versions matching "`2.1.0`"...
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/aws v3.62.0
- Installing hashicorp/http v2.1.0...
- Installed hashicorp/http v2.1.0 (signed by HashiCorp)
- Using previously-installed hashicorp/random v3.0.0

Terraform has made some changes to the provider dependency selections recorded

in the `.terraform.lock.hcl` file. Review those changes and commit them to your version control system if they represent changes you intended to make.

Terraform has been successfully initialized!

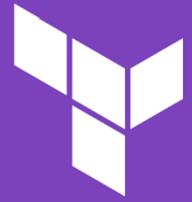
You may now begin working with Terraform. Try running "`terraform plan`" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

```
terraform version
```

```
Terraform v1.0.8
```





```
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.0.0
```

### Task 3: Install Terraform Random provider version

The “random” provider allows the use of randomness within Terraform configurations. This is a logical provider, which means that it works entirely within Terraform’s logic, and doesn’t interact with any other services.

Some previous labs may have already included installing this provider. This can be validate by running a `terraform version` and checking if the `+ provider registry.terraform.io/hashicorp/random v3.0.0` is present.

If it is not update the `terraform.tf` to include the provider.

```
terraform .tf
```

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
    }
    http = {
      source = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source = "hashicorp/random"
      version = "3.1.0"
    }
  }
}
```

Run a `terraform init -upgrade` to validate you pull down the provider versions specified in the configuration and validate with a `terraform version` or a `terraform providers` command.

```
terraform init -upgrade
```

```
terraform version
```

```
Terraform v1.0.8
on linux_amd64
```





```
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

`terraform providers`

Providers required by configuration:

```
-- provider[registry.terraform.io/hashicorp/http] 2.1.0
-- provider[registry.terraform.io/hashicorp/random] 3.1.0
-- provider[registry.terraform.io/hashicorp/aws]
```

Providers required by state:

```
provider[registry.terraform.io/hashicorp/aws]
provider[registry.terraform.io/hashicorp/random]
```

## Task 5: Install Terraform Local provider

The Local provider is used to manage local resources, such as files.

`terraform.tf`

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
  }
}
```

`terraform init -upgrade`





```
`terraform verson
```

```
Terraform v1.0.8  
on linux_amd64  
+ provider registry.terraform.io/hashicorp/aws v3.62.0  
+ provider registry.terraform.io/hashicorp/http v2.1.0  
+ provider registry.terraform.io/hashicorp/local v2.1.0  
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

```
terraform providers
```

```
Providers required by configuration:
```

```
-- provider[registry.terraform.io/hashicorp/http] 2.1.0  
-- provider[registry.terraform.io/hashicorp/random] 3.1.0  
-- provider[registry.terraform.io/hashicorp/local] 2.1.0  
-- provider[registry.terraform.io/hashicorp/aws]
```

```
Providers required by state:
```

```
provider[registry.terraform.io/hashicorp/aws]  
provider[registry.terraform.io/hashicorp/random]
```

As you can see it is very easy to install multiple providers within a single Terraform configuration.





## Lab: Generate SSH Key with Terraform TLS Provider

The Terraform TLS provider provides utilities for working with Transport Layer Security keys and certificates. It provides resources that allow private keys, certificates and certificate requests to be created as part of a Terraform deployment.

- Task 1: Check Terraform
- Task 2: Install Terraform TLS Provider
- Task 3: Creates a self-signed certificate with TLS Provider

### Task 1: Check Terraform version

Run the following command to check the Terraform version:

```
terraform -version
```

You should see:

```
Terraform v1.0.10
```

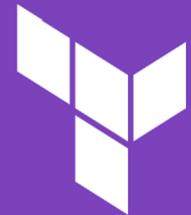
### Task 2: Install Terraform TLS Provider

Edit the file titled `terraform.tf` to add the Terraform TLS provider.

`terraform.tf`

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source = "hashicorp/local"
      version = "2.1.0"
    }
  }
}
```





```
    }
  tls = {
    source  = "hashicorp/tls"
    version = "3.1.0"
  }
}
```

Install the provider by performing a `terraform init`

```
terraform init
```

This informs Terraform that it will install the TLS provider for working with Transport Layer Security keys and certificates.

Validate the installation by running a `terraform version`

terraform version

```
Terraform v1.0.10
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

### **Task 3: Creates a self-signed certificate with TLS Provider**

Update the `main.tf` file with the following configuration blocks for generating a TLS self signed certificate and saving the private key locally.

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "MyAWSKey.pem"
}
```

Note: This example creates a self-signed certificate for a development environment. THIS IS NOT RECOMMENDED FOR PRODUCTION SERVICES.

## Create the Keypair via Terraform





```
terraform apply
```

Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

Apply **complete!** Resources: 2 added, 0 changed, 0 destroyed.

Validate you now have a self-signed private key inside your current working directory name `MyAWSKey.pem`

```
ls -la
```

```
-rwxr-xr-x 1 student student MyAWSKey.pem
```

```
cat MyAWSKey.pem
```

```
-----BEGIN RSA PRIVATE KEY-----
```

```
MIEowIBAAKCAQEAv7Y/09hkhorQUPEdsmWSej62NoTucPvZCyaKqtKs/UnGM31A
Q2w7yQ2VPuEUP1SPJsk0CgJFRhdLQ/uUFWJAtTTE0iV2s0KL77mMZKREsmG3wT0
m5Xh3uT8SPhkLF0wgQfPjVdCKFNs8Qip4S6sktWgtw0mV/SioDNYZArNhV+viye0
wa/jXu55H/C1d4HcaALb9+Ucrut0M5Kqo1Xjgt6gGDpQjidK0Qsdjyaoz8JKQMbk
fZV3DWC9RJ+Sm94gqt9QbIjgnCvlp8IWEcyMBA6P99RSjNB/geH/H4Jvoeg7W1B1
56/s1vZ4G0XzydlyLY7v5j0vyRumAqq5boCv3QIDAQABoIBAFv4QoaOuSsSTP2H
rod20t5yV1ewTfNF3snKo5tvli2sxrjMzZeKx00e8IpJ0DzRhBtHSv/CNxixYhor
Bs97Yy+LMSDfeCFDjX5jtUZTw3EP3PQAnJhHPyR/Fcir40KjA3njFV7pDYPRAch
L58nlQKcGY23cT2gzq0r/iuAQzhH6SRoNvu+DXvKPGWIpWxSPhhSHMHe08N3spLJ
4DptEf0hJ/IiBIJpXoaN09sYVBoRUFd4g3SapqP0cwVK27R/Zy1Fc0ujuqzeE/0t
Rxk80aVSa+50F4SnY7LAGtdME3D39d6c4FpVGXKAoOp1tFpswAQDubib64YKDSNU
0/fMLMECgYEAxVtJwK4v7BCNqNLW9cR5A/LLTFGgVpBNP9BIyV0aCrx9rjezC7ny
Q+2I3/dMAPI9+kuchktzPfsTjUEGM01Iu6HLQjZDECOVC4tUyjA41I1Unz2E2t3u
bwXQN5bDVPLX14VH4Gt/8Ku1wLu+u79jmaByT6jHA3S2iNkrQBrr1M8CgYEA+K2T
eIniHzjRsx7PZmrMDN/qIi6VC6zUfwSaoQxmreQ0FlQixNumfjnJ3/wBWvsL1f89
5xG1m7Ro4WA+0dQHr3POHCY/JiAVN7CBVjAUGUA/RjkuDsSujFrGx0UOEgM0RZe
k7uW4BjNiVSp00v7bZXPY23hOsZYMLmqB/b85MCgYB22+npP37hH38RhBmuXqu7
cwh5aFe2iqXbnueXS0BnQuo2eJk+oLiFrJNYv6lokHw/ODaGsv4u//3gfp7rWspJ
JsIxmfh/ac6j60AfnTc82/lxBi3zWuHzyN3u/L+bc74GsOB/Cn89RUysqjXPAQ9N
QNjXo4BoVmwxwsspXi18pBQKBgC9QLw+vBD02hsdSpFkzFpGYhJ5uSHNJNcDY6mab
ymkaLOLWrSrRM7MuYYdZFhTuUMktX+S3zNrMD2xZ+HnJopCyMtPOPx0M4qjrHPUR
dr2VDvZ6pwGaU6zTPDKTbMZshuu9Gs920HTgozJuxif/A95Ms4GSZVjeZecXXeQt
85Y7AoGBALUqqStHLKYwFI2WRiBkHfz2Kx/aP81F+q+ngBMQWm26eQW7TrBPdaj
QVXqbZSdF3IDYF3Whk42QAVKqiimYCGqbZoUokWobtENrHAhtjznH4etqXQLTsJp
q90kP3kpPogLdTClQNz06x4tFx4M5P+GZViynodX+jZfcP0C41VB
```





-----END RSA PRIVATE KEY-----

We will use this private key in a future lab for associating with our server instances and using it for a means of authentication.





## Lab: Fetch, Version and Upgrade Terraform Providers

Terraform relies on plugins called “providers” to interact with remote systems and expand functionality. Terraform providers can be versioned inside a Terraform configuration block. To prevent external changes from causing unintentional changes, it’s highly recommended that providers specify versions which they are tied to. Depending on the level of acceptable risk and management effort to be tracking version updates, that can either be hard locked to a particular version, or use looser mechanisms such as less than next major rev, or using tilde to track through bug fix versions.

- Task 1: Check Terraform and Provider version
- Task 2: Require specific versions of Terraform providers
- Task 3: Upgrade provider versions

### Task 1: Check Terraform version

Run the following command to check the Terraform version:

```
terraform -version
```

You should see:

```
Terraform v1.0.10
```

### Task 2: Require specific versions of Terraform

Update the `terraform.tf` to define a minimum version of the Terraform AWS provider and Terraform Random provider to be used.

Note: Terraform has multi-provider support from a single set of configuration files. This is extremely beneficial and we can control the version parameters for each individual provider.

```
terraform.tf
```

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
}
```





```

random = {
  source  = "hashicorp/random"
  version = "3.1.0"
}
}
}
}

```

Note: You can always find the latest version of a provider on its provider registry page at <https://registry.terraform.io>.

You need to let Terraform know to use the provider through a `required_providers` block. Now that you have told Terraform to use this new provider you will have to run the `init` command. This will cause Terraform to notice the change and download the correct version if it was not already downloaded.

```
terraform init
```

By default Terraform will always pull the latest provider if no version is set. However setting a version provides a way to ensure your Terraform code remains working in the event a newer version introduces a change that would not work with your existing code. To have more strict controls over the version you may want to require a specific version ( e.g. `version = "= 3.0.0"` ) or use the `~>` operator to only allow the right-most version number to increment.

To check the `terraform` version and provider version installed via `terraform init` run the `terraform version` command.

You can modify the version line of the AWS provider to be as specific or general as desired.

Update the `version` line within your `terraform.tf` file to try these different versioning techniques.

```

version = "~> 3.0"
version = ">= 3.0.0, < 3.1.0"
version = ">= 3.0.0, <= 3.1.0"
version = "~> 2.0"
version = "~> 3.0"

```

Notice that for certain combinations of the version constraint you may encounter an error stating the version of the provider installed does not match the configured version.

```
terraform init
```

```
Initializing the backend...
```





```
Initializing provider plugins...
```

- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/random v3.1.0

```
Error: Failed to query available provider packages
```

```
Could not retrieve the list of available versions for provider hashicorp/
aws: locked provider registry.terraform.io/hashicorp/aws 3.62.0 does
not match configured version constraint >= 3.0.0, < 3.1.0; must use
terraform init -upgrade to allow selection of new versions
```

In these situations you have to intentionally install a version that matches the constraint by issuing a `terraform init -upgrade` command.

### Task 3: How to upgrade provider versions

Terraform configurations must declare which providers they require so that Terraform can install and use them. Once installed, Terraform will record the provider versions in a dependency lock file to ensure that others using this configuration will utilize the same Terraform and provider versions. This file by default is saved as a `.terraform.lock.hcl` file, and its contents look as follows:

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version      = "3.61.0"
  constraints = "~> 3.0"
  hashes = [
    "h1:fpZ14qQnn+uE002Z0lBFHgty480l8I0wd+ewxZ4z3zc=",
    "zh:0483ca802ddb0ae4f73144b4357ba72242c6e2641aeb460b1aa9a6f6965464b0",
    "zh:274712214ebeb0c1269cbc468e5705bb5741dc45b05c05e9793ca97f22a1baa1",
    "zh:3c6bd97a2ca809469ae38f6893348386c476cb3065b120b785353c1507401adf",
    "zh:53dd41a9aed9860adbbeeb71a23e4f8195c656fd15a02c90fa2d302a5f577d8c",
    "zh:65c639c547b97bc880fd83e65511c0f4bbfc91b63cada3b8c0d5776444221700",
    "zh:a2769e19137ff480c1dd3e4f248e832df90fb6930a22c66264d9793895161714",
    "zh:a5897a99332cc0071e46a71359b86a8e53ab09c1453e94cd7cf45a0b577ff590",
    "zh:bdc2353642d16d8e2437a9015cd4216a1772be9736645cc17d1a197480e2b5b7",
    "zh:cbeace1deae938f6c0aca3734e6088f3633ca09611aff701c15cb6d42f2b918a",
    "zh:d33ca19012aab98cc03fdecc0bd5ce56e28f61a1dfbb2eea88e89487de7fb3",
    "zh:d548b29a864b0687e85e8a993f208e25e3ecc40fcc5b671e1985754b32fdd658",
  ]
}

provider "registry.terraform.io/hashicorp/random" {
  version      = "3.0.0"
```





## Hands-On Labs

```
constraints = "3.0.0"
hashes = [
  "h1:yHJpb4IfQQfui07qjUXuUFTU/s+ensuEpm23A+VWz0=",
  "zh:0fcbb00ff8b87dcac1b0ee10831e47e0203a6c46aaf76cb140ba2bab81f02c6b",
  "zh:123c984c0e04bad910c421028d18aa2ca4af25a153264aef747521f4e7c36a17",
  "zh:287443bc6fd7fa9a4341dec235589293cbcc6e467a042ae225fd5d161e4e68dc",
  "zh:2c1be5596dd3cca4859466885eaedf0345c8e7628503872610629e275d71b0d2",
  "zh:684a2ef6f415287944a3d966c4c8cee82c20e393e096e2f7cdcb4b2528407f6b",
  "zh:7625ccb6ff17c2d5360ff2af7f9261c3f213765642dcd84e84ae02a3768fd51",
  "zh:9a60811ab9e6a5bfa6352fbb943bb530acb6198282a49373283a8fa3aa2b43fc",
  "zh:c73e0eaeee6c65b1cf5098b101d51a2789b054201ce7986a6d206a9e2dacaef",
  "zh:e8f9ed41ac83dbe407de9f0206ef1148204a0d51ba240318af801ffb3ee5f578",
  "zh:fbdd0684e62563d3ac33425b0ac9439d543a3942465f4b26582bcfabcb149515",
]
}
```

By default, if a `.terraform.lock.hcl` file exists within a Terraform working directory, the provider versions specified in this file will be used. This lock file helps to ensure that runs across teams will be consistent. As new versions of Terraform providers are released it is often beneficial to upgrade provider versions to take advantage of these updates. This can be accomplished by updating our configuration definition to the desired provider version and running an upgrade.

In this case we are going to update the Terraform random provider from `3.0.0` to `3.1.0`.

Note: If the provider is already at version `3.1.0` then you can downgrade it to `3.0.0` following similar steps.

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
  }
}
```

```
terraform init -upgrade
```

Note: You should never directly modify the lock file.

Once complete you can open the `.terraform.lock.hcl` to see the provider updates, as well as run





## Hands-On Labs

a `terraform version` to see that the new provider version has been installed and ready for use.

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version      = "3.61.0"
  constraints = "~> 3.0"
  hashes = [
    "h1:fpZ14qQnn+uE002Z0lBFHgty480l8I0wd+ewxZ4z3zc=",
    "zh:0483ca802ddb0ae4f73144b4357ba72242c6e2641aeb460b1aa9a6f6965464b0",
    "zh:274712214ebeb0c1269cbc468e5705bb5741dc45b05c05e9793ca97f22a1baa1",
    "zh:3c6bd97a2ca809469ae38f6893348386c476cb3065b120b785353c1507401adf",
    "zh:53dd41a9aed9860adbbeeb71a23e4f8195c656fd15a02c90fa2d302a5f577d8c",
    "zh:65c639c547b97bc880fd83e65511c0f4bbfc91b63cada3b8c0d5776444221700",
    "zh:a2769e19137ff480c1dd3e4f248e832df90fb6930a22c66264d9793895161714",
    "zh:a5897a99332cc0071e46a71359b86a8e53ab09c1453e94cd7cf45a0b577ff590",
    "zh:bdc2353642d16d8e2437a9015cd4216a1772be9736645cc17d1a197480e2b5b7",
    "zh:cbeace1deae938f6c0aca3734e6088f3633ca09611aff701c15cb6d42f2b918a",
    "zh:d33ca19012aab98cc03fdecc0bd5ce56e28f61a1dfbb2eea88e89487de7fb3",
    "zh:d548b29a864b0687e85e8a993f208e25e3ecc40fcc5b671e1985754b32fdd658",
  ]
}

provider "registry.terraform.io/hashicorp/random" {
  version      = "3.1.0"
  constraints = "3.1.0"
  hashes = [
    "h1:rKYu5ZUbXwrLG1w81k7H3nce/Ys6yAxXhWcbtk36HjY=",
    "zh:2bbb3339f0643b5daa07480ef4397bd23a79963cc364cdfbb4e86354cb7725bc",
    "zh:3cd456047805bf639fbf2c761b1848880ea703a054f76db51852008b11008626",
    "zh:4f251b0eda5bb5e3dc26ea4400dba200018213654b69b4a5f96abee815b4f5ff",
    "zh:7011332745ea061e517fe1319bd6c75054a314155cb2c1199a5b01fe1889a7e2",
    "zh:738ed82858317ccc246691c8b85995bc125ac3b4143043219bd0437adc56c992",
    "zh:7dbe52fac7bb21227acd7529b487511c91f4107db9cc4414f50d04ffc3cab427",
    "zh:a3a9251fb15f93e4fcf1789800fc2d7414bbc18944ad4c5c98f466e6477c42bc",
    "zh:a543ec1a3a8c20635cf374110bd2f87c07374cf2c50617eee2c669b3ceeeaa9f",
    "zh:d9ab41d556a48bd7059f0810cf020500635bfc696c9fc3adab5ea8915c1d886b",
    "zh:d9e13427a7d011dbd654e591b0337e6074eeff8c3b9bb11b2e39eaaf257044fd7",
    "zh:f7605bd1437752114baf601bdf6931debe6dc6bfe3006eb7e9bb9080931dca8a",
  ]
}
```

`terraform version`

Terraform v1.0.10

- + provider registry.terraform.io/hashicorp/aws v3.61.0
- + provider registry.terraform.io/hashicorp/random v3.1.0





## Reference

Terraform Versioning Terraform Dependency Lock File





## Lab: Terraform Provisioners

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

To this point the EC2 web server we have created is useless. We created a server without any running code with no useful services are running on it.

We will utilize Terraform provisioners to deploy a webapp onto the instance we've created. In order run these steps Terraform needs a connection block along with our generated SSH key from the previous labs in order to authenticate into our instance. Terraform can utilize both the `local-exec` provisioner to run commands on our local workstation, and the `remote-exec` provisioner to install security updates along with our web application.

- Task 1: Upload your SSH keypair to AWS and associate to your instance.
- Task 2: Create a Security Group that allows SSH to your instance.
- Task 3: Create a connection block using your SSH keypair.
- Task 4: Use the `local-exec` provisioner to change permissions on your local SSH Key
- Task 5: Create a `remote-exec` provisioner block to pull down and install web application.
- Task 6: Apply your configuration and watch for the remote connection.
- Task 7: Pull up the web application and ssh into the web server (optional)

### Task 1: Create an SSH keypair and associate it to your instance.

In `main.tf` add the following resource blocks to create a key pair in AWS that is associated with your generated key from the previous lab.

```
resource "aws_key_pair" "generated" {
  key_name    = "MyAWSKey"
  public_key  = tls_private_key.generated.public_key_openssh

  lifecycle {
    ignore_changes = [key_name]
  }
}
```

```
terraform apply
```





## Task 2: Create a Security Group that allows SSH to your instance.

### Step 7.1.1

In `main.tf` add the following resource block to create a Security Group that allows SSH access.

```
# Security Groups

resource "aws_security_group" "ingress-ssh" {
  name      = "allow-all-ssh"
  vpc_id    = aws_vpc.vpc.id
  ingress  {
    cidr_blocks = [
      "0.0.0.0/0"
    ]
    from_port   = 22
    to_port     = 22
    protocol   = "tcp"
  }
  // Terraform removes the default rule
  egress  {
    from_port   = 0
    to_port     = 0
    protocol   = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

In `main.tf` add the following resource block to create a Security Group that allows web traffic over the standard HTTP and HTTPS ports.

```
# Create Security Group - Web Traffic
resource "aws_security_group" "vpc-web" {
  name      = "vpc-web-${terraform.workspace}"
  vpc_id    = aws_vpc.vpc.id
  description = "Web Traffic"
  ingress  {
    description = "Allow Port 80"
    from_port   = 80
    to_port     = 80
    protocol   = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress  {
    description = "Allow Port 443"
    from_port   = 443
    to_port     = 443
  }
}
```





```

    protocol      = "tcp"
    cidr_blocks  = ["0.0.0.0/0"]
}

egress {
  description = "Allow all ip and ports outbound"
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}
}

resource "aws_security_group" "vpc-ping" {
  name          = "vpc-ping"
  vpc_id        = aws_vpc.vpc.id
  description   = "ICMP for Ping Access"
  ingress {
    description = "Allow ICMP Traffic"
    from_port   = -1
    to_port     = -1
    protocol    = "icmp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    description = "Allow all ip and ports outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
}

```

### Task 3: Create a connection block using your keypair module outputs.

Replace the `aws_instance` “`ubuntu_server`” resource block in your `main.tf` with the code below to deploy and Ubuntu server, associate the AWS Key, Security Group and connection block for Terraform to connect to your instance:

```

resource "aws_instance" "ubuntu_server" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type                = "t2.micro"
  subnet_id                    = aws_subnet.public_subnets["public_subnet_1"]
                                ".id"
  security_groups              = [aws_security_group.vpc-ping.id,
                                  aws_security_group.ingress-ssh.id, aws_security_group.vpc-web.id]
}

```





```

associate_public_ip_address = true
key_name                  = aws_key_pair.generated.key_name
connection {
  user        = "ubuntu"
  private_key = tls_private_key.generated.private_key_pem
  host        = self.public_ip
}

tags = {
  Name = "Ubuntu EC2 Server"
}

lifecycle {
  ignore_changes = [security_groups]
}

}

```

You will notice that we are referencing other resource blocks via Terraform interpolation syntax to associate the security group, keypair and private key for the connection to our instance. The value of `self` refers to the resource defined by the current block. So `self.public_ip` refers to the public IP address of our `aws_instance.web`.

#### **Task 4: Use the local-exec provisioner to change permissions on your local SSH Key**

The `local-exec` provisioner invokes a local executable after a resource is created. We will utilize a `local-exec` provisioner to make sure our private key is permissioned correctly. This invokes a process on the machine running Terraform, not on the resource.

Update the `aws_instance` “`ubuntu_server`” resource block in your `main.tf` to call a `local-exec` provisioner:

```

# Leave the first part of the block unchanged and create our `local-exec` provisioner
provisioner "local-exec" {
  command = "chmod 600 ${local_file.private_key_pem.filename}"
}

```

#### **Task 5: Create a remote-exec provisioner block to pull down web application.**

The `remote-exec` provisioner runs remote commands on the instance provisioned with Terraform. We can use this provisioner to clone our web application code to the instance and then invoke the





setup script.

```
provisioner "remote-exec" {
  inline = [
    "sudo rm -rf /tmp",
    "sudo git clone https://github.com/hashicorp/demo-terraform-101 /tmp",
    "",
    "sudo sh /tmp/assets/setup-web.sh",
  ]
}
```

Make sure both provisioners are inside the `aws_instance` resource block.

### Task 3: Apply your configuration and watch for the remote connection.

In order to create our security group, new web ubuntu instance with the associated public SSH Key and execute our `provisioners` we will validate our code and then initiate a `terraform apply`.

```
terraform validate
```

```
Success! The configuration is valid.
```

```
terraform apply
```

Upon running `terraform apply`, you should see new output which includes a connection to the EC2 instance

```
terraform apply
```

```
...
aws_instance.ubuntu_server: Provisioning with 'local-exec'...
aws_instance.ubuntu_server (local-exec): Executing: ["/bin/sh" "-c" "chmod
600 MyAWSKey.pem"]
aws_instance.ubuntu_server: Provisioning with 'remote-exec'...
aws_instance.ubuntu_server (remote-exec): Connecting to remote host via
SSH...
aws_instance.ubuntu_server (remote-exec): Host: 3.236.92.141
aws_instance.ubuntu_server (remote-exec): User: ubuntu
aws_instance.ubuntu_server (remote-exec): Password: false
aws_instance.ubuntu_server (remote-exec): Private key: true
aws_instance.ubuntu_server (remote-exec): Certificate: false
aws_instance.ubuntu_server (remote-exec): SSH Agent: true
aws_instance.ubuntu_server (remote-exec): Checking Host Key: false
aws_instance.ubuntu_server (remote-exec): Target Platform: unix
aws_instance.ubuntu_server (remote-exec): Connecting to remote host via
SSH...
```





## Hands-On Labs

```

aws_instance.ubuntu_server (remote-exec): Host: 3.236.92.141
aws_instance.ubuntu_server (remote-exec): User: ubuntu
aws_instance.ubuntu_server (remote-exec): Password: false
aws_instance.ubuntu_server (remote-exec): Private key: true
aws_instance.ubuntu_server (remote-exec): Certificate: false
aws_instance.ubuntu_server (remote-exec): SSH Agent: true
aws_instance.ubuntu_server (remote-exec): Checking Host Key: false
aws_instance.ubuntu_server (remote-exec): Target Platform: unix
aws_instance.ubuntu_server (remote-exec): Connecting to remote host via
    SSH...
aws_instance.ubuntu_server (remote-exec): Host: 3.236.92.141
aws_instance.ubuntu_server (remote-exec): User: ubuntu
aws_instance.ubuntu_server (remote-exec): Password: false
aws_instance.ubuntu_server (remote-exec): Private key: true
aws_instance.ubuntu_server (remote-exec): Certificate: false
aws_instance.ubuntu_server (remote-exec): SSH Agent: true
aws_instance.ubuntu_server (remote-exec): Checking Host Key: false
aws_instance.ubuntu_server (remote-exec): Target Platform: unix
aws_instance.ubuntu_server (remote-exec): Connected!
aws_instance.ubuntu_server: Still creating... [50s elapsed]
aws_instance.ubuntu_server (remote-exec): Cloning into '/tmp'...
aws_instance.ubuntu_server (remote-exec): remote: Enumerating objects:
    417, done.
aws_instance.ubuntu_server (remote-exec): Receiving objects:  0% (1/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  1% (5/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  2% (9/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  3% (13/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  4% (17/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  5% (21/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  6% (26/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  7% (30/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  8% (34/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects:  9% (38/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 10% (42/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 11% (46/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 12% (51/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 13% (55/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 14% (59/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 15% (63/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 16% (67/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 17% (71/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 18% (76/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 19% (80/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 20% (84/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 21% (88/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 22% (92/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 23% (96/417)
aws_instance.ubuntu_server (remote-exec): Receiving objects: 24%
    (101/417), 2.71 MiB | 5.37 MiB/s

```





## Hands-On Labs

```

aws_instance.ubuntu_server (remote-exec): Receiving objects: 25%
(105/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 26%
(109/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 27%
(113/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 28%
(117/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 29%
(121/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 30%
(126/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 31%
(130/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 32%
(134/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 33%
(138/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 34%
(142/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 35%
(146/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 36%
(151/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 37%
(155/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 38%
(159/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 39%
(163/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 40%
(167/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 41%
(171/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 42%
(176/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 43%
(180/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 44%
(184/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 45%
(188/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 46%
(192/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 47%
(196/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): remote: Total 417 (delta 0),
reused 0 (delta 0), pack-reused 417
aws_instance.ubuntu_server (remote-exec): Receiving objects: 48%

```





## Hands-On Labs

```
(201/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 49%
(205/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 50%
(209/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 51%
(213/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 52%
(217/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 53%
(222/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 54%
(226/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 55%
(230/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 56%
(234/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 57%
(238/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 58%
(242/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 59%
(247/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 60%
(251/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 61%
(255/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 62%
(259/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 63%
(263/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 64%
(267/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 65%
(272/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 66%
(276/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 67%
(280/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 68%
(284/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 69%
(288/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 70%
(292/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 71%
(297/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 72%
(301/417), 2.71 MiB | 5.37 MiB/s
```





## Hands-On Labs

```

aws_instance.ubuntu_server (remote-exec): Receiving objects: 73%
  (305/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 74%
  (309/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 75%
  (313/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 76%
  (317/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 77%
  (322/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 78%
  (326/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 79%
  (330/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 80%
  (334/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 81%
  (338/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 82%
  (342/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 83%
  (347/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 84%
  (351/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 85%
  (355/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 86%
  (359/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 87%
  (363/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 88%
  (367/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 89%
  (372/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 90%
  (376/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 91%
  (380/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 92%
  (384/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 93%
  (388/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 94%
  (392/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 95%
  (397/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 96%
  (401/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 97%

```





## Hands-On Labs

```
(405/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 98%
(409/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 99%
(413/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 100%
(417/417), 2.71 MiB | 5.37 MiB/s
aws_instance.ubuntu_server (remote-exec): Receiving objects: 100%
(417/417), 4.18 MiB | 5.76 MiB/s, done.
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 0% (0/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 2% (4/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 3% (5/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 4% (6/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 5% (8/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 6% (9/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 7% (11/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 12% (18/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 13% (19/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 14% (20/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 15% (22/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 16% (23/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 17% (25/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 21% (31/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 25% (36/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 26% (37/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 30% (43/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 31% (45/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 34% (49/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 35% (50/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 39% (56/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 41% (59/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 42% (60/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 44% (63/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 48% (69/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 49% (70/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 63% (90/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 64% (91/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 66% (95/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 68% (97/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 69% (98/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 76% (108/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 80% (115/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 81% (116/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 82% (117/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 84% (120/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 88% (125/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 94% (134/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 95% (135/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 97% (139/142)
```





```

aws_instance.ubuntu_server (remote-exec): Resolving deltas: 100% (142/142)
aws_instance.ubuntu_server (remote-exec): Resolving deltas: 100% (142/142)
    , done.
aws_instance.ubuntu_server (remote-exec): Created symlink /etc/systemd/
    system/multi-user.target.wants/webapp.service -> /lib/systemd/system/
    webapp.service.
aws_instance.ubuntu_server: Creation complete after 54s [id=i-021
    cf7ae83ee067d1]
...

```

### Task 7: Pull up the web application and ssh into the web server (optional)

You can now visit your web application by pointing your browser at the public\_ip output for your EC2 instance. To get that address you can look at the state details of the EC2 instance by performing a `terraform state show aws_instance.ubuntu_server`

```
terraform state show aws_instance.ubuntu_server
```

```

resource "aws_instance" "ubuntu_server" {
  ami                               = "ami-0964546d3da97e3ab"
  arn                             = "arn:aws:ec2:us-west-
  -2:508140242758:instance/i-00eccad2a464a4aa3"
  associate_public_ip_address      = true
  availability_zone                 = "us-west-2b"
  cpu_core_count                   = 1
  cpu_threads_per_core            = 1
  disable_api_termination        = false
  ebs_optimized                    = false
  get_password_data               = false
  hibernation                      = false
  id                                = "i-00eccad2a464a4aa3"
  instance_initiated_shutdown_behavior = "stop"
  instance_state                   = "running"
  instance_type                     = "t2.micro"
  ipv6_address_count              = 0
  ipv6_addresses                   = []
  key_name                          = "MyAWSKey"
  monitoring                        = false
  primary_network_interface_id     = "eni-00e236032e4f38e95"
  private_dns                       = "ip-10-0-101-238.us-west-2.
  compute.internal"
  private_ip                        = "10.0.101.238"
  public_ip                         = "35.86.144.200"
  secondary_private_ips            = []
  security_groups                  = [
    "sg-068db6e720cb80a46",

```





```

    "sg-0dbb6b4429d7730f2",
    "sg-0f64195ac2bfee1f2",
]
source_dest_check          = true
subnet_id                  = "subnet-03977b3f439ccc2cb"
tags                        = {
    "Name" = "Ubuntu EC2 Server"
}
tags_all                   = {
    "Name"      = "Ubuntu EC2 Server"
    "Owner"     = "Acme"
    "Provisioned" = "Terraform"
}
tenancy                     = "default"
vpc_security_group_ids     = [
    "sg-068db6e720cb80a46",
    "sg-0dbb6b4429d7730f2",
    "sg-0f64195ac2bfee1f2",
]
capacity_reservation_specification {
    capacity_reservation_preference = "open"
}

credit_specification {
    cpu_credits = "standard"
}

enclave_options {
    enabled = false
}

metadata_options {
    http_endpoint           = "enabled"
    http_put_response_hop_limit = 1
    http_tokens             = "optional"
}

root_block_device {
    delete_on_termination = true
    device_name           = "/dev/sda1"
    encrypted              = false
    iops                   = 100
    tags                   = {}
    throughput              = 0
    volume_id               = "vol-06c0a4100fe14914c"
    volume_size              = 8
    volume_type              = "gp2"
}

```



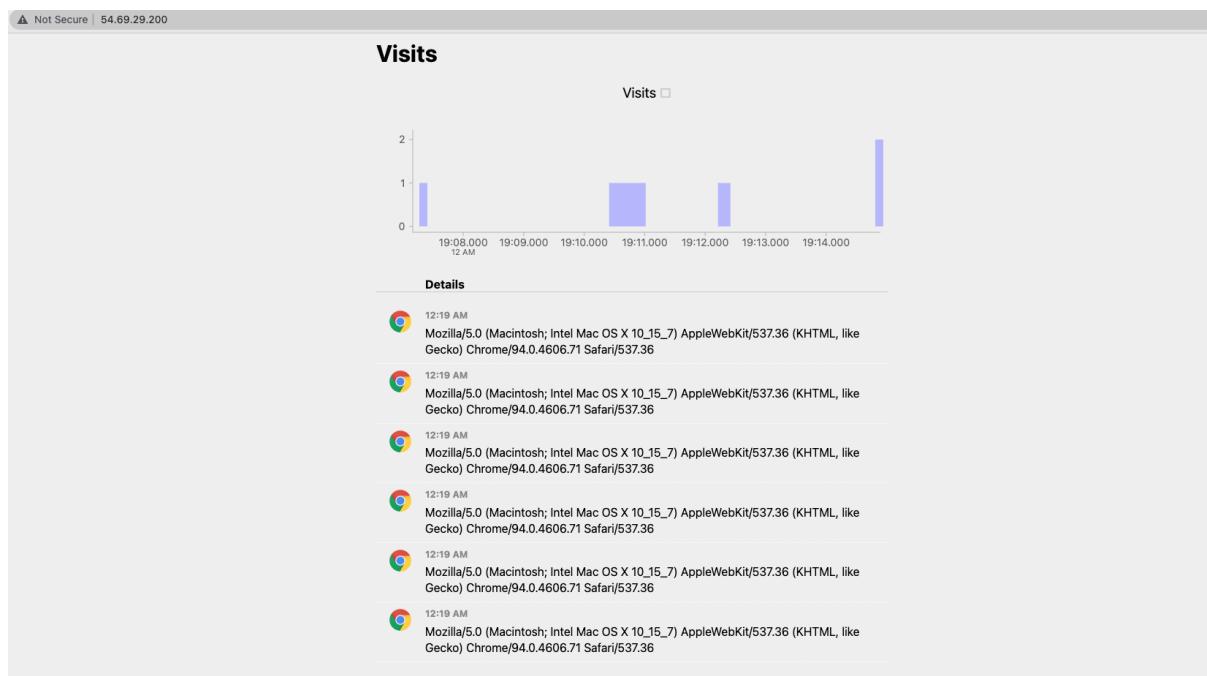
# HashiCorp Certified: Terraform Associate

## Hands-On Labs



```
}
```

Visit [http://<public\\_ip>](http://<public_ip>)



**Figure 1:** Web Application

### Optional

If you want, you can also ssh to your EC2 instance with a command like `ssh -i MyAWSKey.pem ubuntu@<public_ip>`. Type yes when prompted to use the key. Type `exit` to quit the ssh session.

```
ssh -i MyAWSKey.pem ubuntu@<public_ip>
```

```
ssh -i MyAWSKey.pem ubuntu@54.69.29.200
The authenticity of host '54.69.29.200 (54.69.29.200)' can't be
established.
ECDSA key fingerprint is SHA256:
0gKh9TuNNuyFFBT96oizbYTGhvtZKAoLOIcFgLw7Niw.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '54.69.29.200' (ECDSA) to the list of known
hosts.
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1019-aws x86_64)

 * Documentation:  https://help.ubuntu.com
```





```
* Management:      https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
```

```
System information as of Wed Oct 13 04:22:02 UTC 2021
```

```
System load: 0.0          Processes:      102
Usage of /: 18.4% of 7.69GB  Users logged in: 0
Memory usage: 20%          IPv4 address for eth0: 10.0.101.134
Swap usage: 0%
```

```
1 update can be applied immediately.
To see these additional updates run: apt list --upgradable
```

```
The list of available updates is more than a week old.
To check for new updates run: sudo apt update
```

```
Last login: Wed Oct 13 04:17:21 2021 from 44.197.238.120
ubuntu@ip-10-0-101-134:~$ exit
logout
Connection to 54.69.29.200 closed.
```





## Lab: Auto Formatting Terraform Code

Terraform provides a useful subcommand that can be used to easily format all of your code. This subcommand is called `fmt`. The subcommand allows you to easily format your Terraform code to a canonical format and style based on a subset of Terraform language style conventions.

### Task 1: Format misaligned Terraform Code

Observe the code below, with the incorrect spacing and misaligned equals signs. Although this code is technically valid, it looks messy. Instead of manually fixing this, you can use a `fmt` to fix it. Update the following resource block in your `main.tf` file:

`main.tf`

```
resource "random_string" "random" {
    length = 10
    special = true
    min_numeric = 6
        min_special = 2
    min_upper = 3
}
```

To fix it, run the `fmt` subcommand as shown below:

```
terraform fmt
```

Looking at the same `main.tf` file now, you will see the `fmt` subcommand aligned the equal signs!

```
resource "random_string" "random" {
    length      = 10
    special     = true
    min_numeric = 6
    min_special = 2
    min_upper   = 3
}
```

The `terraform fmt` command is very useful to use before checking code into version control, and many will include it as a `pre-commit` hook before checking their code into a version control system.





## Lab: Terraform Taint and Replace

The terraform `taint` command allows for you to manually mark a resource for recreation. The `taint` command informs Terraform that a particular resource needs to be rebuilt even if there has been no configuration change for that resource. Terraform represents this by marking the resource as “tainted” in the Terraform state, in which case Terraform will propose to replace it in the next plan you create.

- Task 1: Manually mark a resource to be rebuilt by using the terraform `taint` command.
- Task 2: Observe a `remote-exec` provisioner failing, resulting in Terraform automatically tainting a resource.
- Task 3: Untaint a resource
- Task 4: Use the `-replace` option rather than `taint`

### Task 1: Manually mark a resource to be rebuilt by using the terraform `taint` command

There are some situations where we want to recreate a resource without modifying any terraform configuration. An example of this might be to rebuild a server and force it to rerun its bootstrap process. This can be accomplished by manually “tainting” the resource.

#### Step 1.1 - Create a new Web Server

Let's add a new webserver to our configuration that we can work on. Update your `main.tf` to include the new webserver.

```
# Terraform Resource Block - To Build Web Server in Public Subnet
resource "aws_instance" "web_server" {
  ami                               = data.aws_ami.ubuntu.id
  instance_type                     = "t2.micro"
  subnet_id                         = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups                   = [aws_security_group.vpc-ping.id,
                                         aws_security_group.ingress-ssh.id, aws_security_group.vpc-web.id]
  associate_public_ip_address       = true
  key_name                           = aws_key_pair.generated.key_name
  connection {
    user     = "ubuntu"
    private_key = tls_private_key.generated.private_key_pem
    host     = self.public_ip
  }
}
```





```
# Leave the first part of the block unchanged and create our `local-exec` provisioner
provisioner "local-exec" {
  command = "chmod 600 ${local_file.private_key_pem.filename}"
}

provisioner "remote-exec" {
  inline = [
    "sudo rm -rf /tmp",
    "sudo git clone https://github.com/hashicorp/demo-terraform-101 /tmp",
    "sudo sh /tmp/assets/setup-web.sh",
  ]
}

tags = {
  Name = "Web EC2 Server"
}

lifecycle {
  ignore_changes = [security_groups]
}

}
```

Execute a `plan` and `apply` to build out the web server.

```
terraform validate
terraform plan
terraform apply
```

### Step 1.2 - Reference the Web Server

Now that that server has been built, let's validate it's resource ID. If you're unsure of the correct ID for the resource you want to taint, you can use `terraform state list`:

```
terraform state list
```

```
data.aws_ami.ubuntu
data.aws_ami.ubuntu_16_04
data.aws_ami.windows_2019
data.aws_availability_zones.available
data.aws_region.current
aws_eip.nat_gateway_eip
aws_instance.ubuntu_server
aws_instance.web_server
```





```
aws_internet_gateway.internet_gateway
aws_key_pair.generated
aws_nat_gateway.nat_gateway
aws_route_table.private_route_table
aws_route_table.public_route_table
aws_route_table_association.private["private_subnet_1"]
aws_route_table_association.private["private_subnet_2"]
aws_route_table_association.private["private_subnet_3"]
aws_route_table_association.public["public_subnet_1"]
aws_route_table_association.public["public_subnet_2"]
aws_route_table_association.public["public_subnet_3"]
aws_security_group.ingress-ssh
aws_security_group.vpc-ping
aws_security_group.vpc-web
aws_subnet.private_subnets["private_subnet_1"]
aws_subnet.private_subnets["private_subnet_2"]
aws_subnet.private_subnets["private_subnet_3"]
aws_subnet.public_subnets["public_subnet_1"]
aws_subnet.public_subnets["public_subnet_2"]
aws_subnet.public_subnets["public_subnet_3"]
aws_vpc.vpc
local_file.private_key_pem
tls_private_key.generated
```

### Step 1.3 - Mark the webserver to be recreated with the taint command

We can force the resource to be destroyed and recreated with the taint command:

```
terraform taint aws_instance.web_server
Resource instance aws_instance.web_server has been marked as tainted.
```

### Step 1.4 - Recreate the Web Server

Now we execute a `terraform plan` and `terraform apply` we will notice that terraform has marked the resource to be recreated. This is because the resource has been tainted.

```
terraform plan

# aws_instance.web_server is tainted, so must be replaced
-/+ resource "aws_instance" "web_server" {

...
Plan: 1 to add, 0 to change, 1 to destroy.
```





```
terraform apply
```

```
Plan: 1 to add, 0 to change, 1 to destroy.
```

```
Do you want to perform these actions?
```

```
Terraform will perform the actions described above.
```

```
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
aws_instance.web_server: Destroying... [id=i-0c06f6a2b91486226]
aws_instance.web_server: Still destroying... [id=i-0c06f6a2b91486226, 10s
  elapsed]
aws_instance.web_server: Still destroying... [id=i-0c06f6a2b91486226, 20s
  elapsed]
aws_instance.web_server: Still destroying... [id=i-0c06f6a2b91486226, 30s
  elapsed]
aws_instance.web_server: Still destroying... [id=i-0c06f6a2b91486226, 40s
  elapsed]
aws_instance.web_server: Destruction complete after 41s
aws_instance.web_server: Creating...
aws_instance.web_server: Still creating... [10s elapsed]
aws_instance.web_server: Still creating... [20s elapsed]
aws_instance.web_server: Still creating... [30s elapsed]
aws_instance.web_server: Provisioning with 'local-exec'...
aws_instance.web_server (local-exec): Executing: ["/bin/sh" "-c" "chmod
  600 MyAWSKey.pem"]
aws_instance.web_server: Provisioning with 'remote-exec'...
aws_instance.web_server (remote-exec): Connecting to remote host via SSH
...
aws_instance.web_server (remote-exec): Host: 18.236.86.172
aws_instance.web_server (remote-exec): User: ubuntu
aws_instance.web_server (remote-exec): Password: false
aws_instance.web_server (remote-exec): Private key: true
aws_instance.web_server (remote-exec): Certificate: false
aws_instance.web_server (remote-exec): SSH Agent: false
aws_instance.web_server (remote-exec): Checking Host Key: false
aws_instance.web_server (remote-exec): Target Platform: unix
...
aws_instance.web_server: Creation complete after 57s [id=i-00
  e3dafde418bdf73]
```





## Task 2: Observe a `remote-exec` provisioner failing, resulting in Terraform automatically tainting a resource

In addition to being able to `taint` a resource manually, Terraform will automatically taint a resource if it is configured with a creation-time provisioner and that provisioner fails. A tainted resource will be planned for destruction and recreation upon the next `terraform apply`. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it.

Modify the `remote-exec` provisioner within the `aws_instance.web_server` resource block to intentionally include a bad command.

```
resource "aws_instance" "web_server" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.
    ingress-ssh.id, aws_security_group.vpc-web.id]
  key_name       = aws_key_pair.generated.key_name
  connection {
    user          = "ubuntu"
    private_key   = tls_private_key.generated.private_key_pem
    host          = self.public_ip
  }
  associate_public_ip_address = true
  tags = {
    Name = "Web EC2 Server"
  }

  provisioner "local-exec" {
    command = "chmod 600 ${local_file.private_key_pem.filename}"
  }

  provisioner "remote-exec" {
    inline = [
      "exit 2",
      "git clone https://github.com/hashicorp/demo-terraform-101",
      "cp -a demo-terraform-101/. /tmp/",
      "sudo sh /tmp/assets/setup-web.sh",
    ]
  }
}
```

Manually `taint` the resource and run a `terraform apply` and observe that the `remote-exec` provisioner fails.





```
terraform taint aws_instance.web_server
terraform apply
```

```
Error: remote-exec provisioner error

  with aws_instance.web_server,
  on main.tf line 169, in resource "aws_instance" "web_server":
169:   provisioner "remote-exec" {

  error executing "/tmp/terraform_1714387465.sh": Process exited with
  status 2
```

This time Terraform automatically tainted the resource. You can see resource has been marked as [tainted](#) by looking at the resource using the `terraform show` command.

```
terraform state show aws_instance.web_server
```

```
# aws_instance.web_server: (tainted)
resource "aws_instance" "web_server" {
```

You will notice that because the `remote-exec` provisioner failed, Terraform automatically marked it as [tainted](#). This informs Terraform that this resource needs to be rebuilt upon the next `terraform apply`.

## Untaint a resource

You can also [untaint](#) a resource by using the `terraform untaint` command.

```
terraform untaint aws_instance.web_server
```

```
Resource instance aws_instance.web_server has been successfully untainted.
```

This informs Terraform that this resource does not need to be rebuilt upon the next `terraform apply`.

## Task 4: Use the `-replace` option rather than `taint`

As of Terraform v0.15.2 and later the `taint` command is deprecated, because there are better alternatives available.

If your intent is to force replacement of a particular object even though there are no configuration changes that would require it, it is recommended to use the `-replace` option with `terraform apply` in place of the deprecated `taint` command.





Remove the offending error from the `remote-exec` provisioner and rebuild the web server resource using the `terraform apply -replace` command.

```
provisioner "remote-exec" {
  inline = [
    "git clone https://github.com/hashicorp/demo-terraform-101",
    "cp -a demo-terraform-101/. /tmp/",
    "sudo sh /tmp/assets/setup-web.sh",
  ]
}
```

```
terraform apply -replace="aws_instance.web_server"
```

```
Plan: 1 to add, 0 to change, 1 to destroy.
```

```
Do you want to perform these actions?
```

```
Terraform will perform the actions described above.
```

```
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
aws_instance.web_server: Destroying... [id=i-00f52f68eec42518b]
aws_instance.web_server: Still destroying... [id=i-00f52f68eec42518b, 10s elapsed]
aws_instance.web_server: Still destroying... [id=i-00f52f68eec42518b, 20s elapsed]
aws_instance.web_server: Still destroying... [id=i-00f52f68eec42518b, 30s elapsed]
aws_instance.web_server: Destruction complete after 31s
aws_instance.web_server: Creating...
aws_instance.web_server: Still creating... [10s elapsed]
aws_instance.web_server: Still creating... [20s elapsed]
aws_instance.web_server: Still creating... [30s elapsed]
aws_instance.web_server: Still creating... [40s elapsed]
aws_instance.web_server: Provisioning with 'local-exec'...
aws_instance.web_server (local-exec): Executing: ["./bin/sh" "-c" "chmod 600 MyAWSKey.pem"]
aws_instance.web_server: Provisioning with 'remote-exec'...
aws_instance.web_server (remote-exec): Connecting to remote host via SSH
...
aws_instance.web_server (remote-exec): Host: 34.214.67.54
aws_instance.web_server (remote-exec): User: ubuntu
aws_instance.web_server (remote-exec): Password: false
aws_instance.web_server (remote-exec): Private key: true
aws_instance.web_server (remote-exec): Certificate: false
aws_instance.web_server (remote-exec): SSH Agent: false
aws_instance.web_server (remote-exec): Checking Host Key: false
aws_instance.web_server (remote-exec): Target Platform: unix
```





```
aws_instance.web_server (remote-exec): Connected!
aws_instance.web_server (remote-exec): Cloning into 'demo-terraform-101'
...
aws_instance.web_server: Still creating... [50s elapsed]
aws_instance.web_server (remote-exec): remote: Enumerating objects: 417,
done.
aws_instance.web_server (remote-exec): Receiving objects:  0% (1/417)
aws_instance.web_server (remote-exec): Receiving objects:  1% (5/417)
aws_instance.web_server (remote-exec): Receiving objects:  2% (9/417)
...
aws_instance.web_server (remote-exec): Resolving deltas: 100% (142/142),
done.
aws_instance.web_server (remote-exec): cp: preserving times for '/tmp/.': 
  Operation not permitted
aws_instance.web_server (remote-exec): Created symlink /etc/systemd/system/
  /multi-user.target.wants/webapp.service -> /lib/systemd/system/webapp.
  service.
aws_instance.web_server: Creation complete after 53s [id=i-03
  db5c21e61154d36]
```

Using the `-replace` command to rebuild a resource is the recommended approach moving forward.





## Lab: Terraform Import

We've already seen many benefits of using Terraform to build out our cloud infrastructure. But what if there are existing resources that we'd also like to manage with Terraform?

Enter [terraform import](#).

With minimal coding and effort, we can add our resources to our configuration and bring them into state.

- Task 1: Manually create EC2 (not with Terraform)
- Task 2: Prepare for a Terraform Import
- Task 3: Import the Resource in Terraform

### Task 1: Manually create EC2 (not with Terraform)

Log into AWS and in the EC2 console, select Instances from the left navigation panel in the VPC console. Click the Launch Instances button in the top right of the AWS console.

The screenshot shows the AWS EC2 Instances page. The left sidebar has 'Instances' selected under 'Instances'. The main area shows a table with one row:

Name	Instance ID	Instance state	Instance type	Status check	Alarm st
Web EC2 Server	i-03db5c21e61154d36	Running	t2.micro	2/2 checks passed	No alarm

**Figure 1:** EC2

Choose a Amazon Linux 2 AMI



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

**Figure 2:** Create EC2

Choose [t2.micro](#) for the Instance Type which is Free Tier Eligible.

Select the appropriate VPC and Public Subnet

Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot Instances to take advantage of the lower pricing, assign an access management role to the instance, and more.

Number of Instances: 1

Purchasing option: Request Spot Instances

Network: new VPC

Subnet: new subnet

Auto-assign Public IP: subnet-02cc2d24001872ba1 | public\_subnet\_1 | us-west-2b

Placement group: Add instance to placement group

Capacity Reservation: Open

Domain join directory: No directory

IAM role: None

Shutdown behavior: Stop

Stop - Hibernate behavior: Enable hibernation as an additional stop behavior

Enable termination protection: Protect against accidental termination

Monitoring: Enable CloudWatch detailed monitoring

**Figure 3:** Configure EC2

Launch the EC2 Instance with your [MyAWSKey](#) pair.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

The screenshot shows the AWS CloudFormation Launch Wizard Step 7: Review Instance Launch. The configuration includes:

- AMI Details:** Amazon Linux 2 AMI (HVM), SSD Volume Type - ami-013a129d325529d
- Instance Type:** t2.micro (1 vCPU, 1 GiB Memory)
- Security Groups:** launch-wizard-1 (created 2021-10-13T08:55:23.527Z)
- Key Pair:** MyAWSKey (RSA)
- Launch Instances** button

**Figure 4:** Launch EC2

Note the instance id that AWS has assigned the EC2 Instance

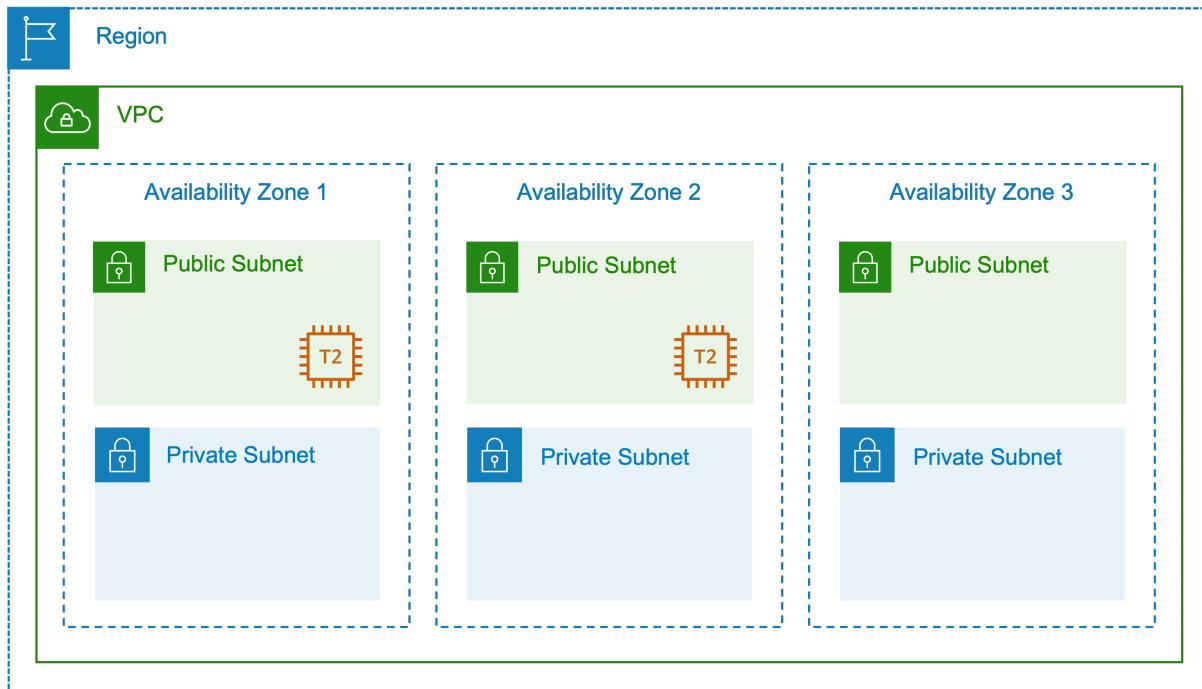
The screenshot shows the AWS CloudWatch Launch Status page with the following content:

- Launch Status:** Your instances are now launching (i-0bfff5070c5fb87b6)
- Get notified of estimated charges:** Create billing alerts to get an email notification when estimated charges on your AWS bill exceed an amount you define (for example, if you exceed the free usage tier).
- How to connect to your instances:** Your instances are launching, and it may take a few minutes until they are in the running state, when they will be ready for you to use. Usage hours on your new instances will start immediately and continue to accrue until you stop or terminate your instances.
- Helpful Resources:**
  - How to connect to your Linux instance
  - Learn about AWS Free Usage Tier
  - Amazon EC2: User Guide
  - Amazon EC2: Discussion Forum

**Figure 5:** EC2 Instance ID

Now our AWS infrastructure diagrams resembles the following were we have one web server created by Terraform in our environment and one web server that has been created manually in our environment (without Terraform)





**Figure 6:** Existing Infrastructure

The objective of the next task will be to import the manually created web server into our Terraform state and configuration so that all of our infrastructure can be managed via code.

## Task 2: Prepare for Import

In order to start the import, our `main.tf` requires a provider configuration. Start off with a provider block with a region in which you manually built the EC2 instance.

```
provider "aws" {
  region = "us-west-2"
}
```

Note: This is most likely already configured in your `main.tf` from previous labs.

You must also have a destination resource to store state against. Add an empty resource block now. We will add an EC2 instance called “aws\_linux”.

```
resource "aws_instance" "aws_linux" {}
```

We’re now all set to import our instance into state!





### Task 3: Import the Resource into Terraform

Using the instance ID provided by your instructor, run the `terraform import` command now. The import command is comprised of four parts.

Example:

- `terraform` to call our binary
- `import` to specify the action to take
- `aws_instance.aws_linux` to specify the resource in our config file (`main.tf`) that this resource corresponds to
- `i-0bfff5070c5fb87b6` to specify the real-world resource (in this case, an AWS EC2 instance) to import into state

**Note:** The resource name and unique identifier of that resource are unique to each configuration.

See what happens below when we've successfully run `terraform import <resource.name> <unique_identifier>`.

The `<unique_identifier>` is the ID you captured at the end of Task 1.

```
terraform import aws_instance.aws_linux i-0bfff5070c5fb87b6
aws_instance.linux: Importing from ID "i-0bfff5070c5fb87b6"...
aws_instance.linux: Import prepared!
  Prepared aws_instance for import
aws_instance.linux: Refreshing state... [id=i-0bfff5070c5fb87b6]

Import successful!
```

The resources that were imported are shown above. These resources are now in your Terraform state and will henceforth be managed by Terraform.

Great! Our resource now exists in our state. But what happens if we were to run a plan against our current config?

```
terraform plan

Error: Missing required argument

  on main.tf line 5, in resource "aws_instance" "linux":
  5: resource "aws_instance" "linux" {

The argument "ami" is required, but no definition was found.
```





Error: Missing required argument

```
on main.tf line 5, in resource "aws_instance" "linux":
  5: resource "aws_instance" "linux" {

The argument "instance_type" is required, but no definition was found.
```

We're missing some required attributes. How can we find those without looking at the console? Think back to our work with the workspace state. What commands will show us the information we need?

We know the exact resource to look for in our state, so let's query it using the `terraform state show` command.

```
terraform state show aws_instance.aws_linux
# aws_instance.aws_linux:
resource "aws_instance" "aws_linux" {
    ami                      = "ami-013a129d325529d4d"
    arn                      = "arn:aws:ec2:us-west
                            -2:508140242758:instance/i-0bfff5070c5fb87b6"
...
}
```

Using the output from the above command, we can now piece together the minimum required attributes for our configuration. Add the required attributes to your resource block and rerun the apply.

```
resource "aws_instance" "aws_linux" {
    ami          = "ami-013a129d325529d4d"
    instance_type = "t2.micro"
}
```

Your `ami` and `instance_type` may differ. Be sure to use the values provided in the previous step.

```
terraform plan
...
# aws_instance.aws_linux will be updated in-place
~ resource "aws_instance" "aws_linux" {
    id                  = "i-0bfff5070c5fb87b6"
    tags                = {}
    ~ tags_all          = {
        + "Owner"      = "Acme"
        + "Provisioned" = "Terraform"
    }
    # (27 unchanged attributes hidden)
    # (6 unchanged blocks hidden)
}
```

You've successfully imported **and** declared your existing resource into your Terraform configuration.





## Hands-On Labs

Notice that Terraform wants to update the default tags for this instance based on the default tags we specified in the Terraform AWS Provider - Default Tags lab.

To apply these default tags you can run a `terraform apply`

```
terraform apply
```

This verifies that you can now modify, update, or destroy this EC2 server using the traditional Terraform configuration and CLI commands now that it has been imported into Terraform.

You can now remove the item from your configuration as this server will no longer be required for future labs.





## Lab: Terraform Workspaces - OSS

Those who adopt Terraform typically want to leverage the principles of DRY (Don't Repeat Yourself) development practices. One way to adopt this principle with respect to IaC is to utilize the same code base for different environments (development, quality, production, etc.)

Workspaces is a Terraform feature that allows us to organize infrastructure by environments and variables in a single directory.

Terraform is based on a stateful architecture and therefore stores state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

The persistent data stored in the state belongs to a Terraform workspace. Initially the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

- Task 1: Using Terraform Workspaces (Open Source)
- Task 2: Create a new Terraform Workspace for Development State
- Task 3: Deploy Infrastructure within the Terraform development workspace
- Task 4: Changing between Workspaces
- Task 5: Utilizing the \${terraform.workspace} interpolation sequence within your configuration

### Task 1: Using Terraform Workspaces (Open Source)

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

You can check the Terraform workspace you are in with the `terraform workspace` command.

```
terraform workspace show  
default
```

To see a list of the options you have within the `terraform workspace` command issue a `terraform workspace -help`

```
terraform workspace -help  
  
Usage: terraform [global options] workspace  
  
    new, list, show, select and delete Terraform workspaces.
```



**Subcommands:**

<code>delete</code>	Delete a workspace
<code>list</code>	List Workspaces
<code>new</code>	Create a new workspace
<code>select</code>	Select a workspace
<code>show</code>	Show the name of the current workspace

We will utilize the current `default` workspace to store the state information for our `us-east-1` aws region.

**Task 2: Create a new Terraform Workspace for Development State**

Leveraging the same code base let's perform a development deployment into the `us-west-2` region while not affecting the infrastructure that was built out in our `default` workspace.

To begin, let's create a new terraform workspace called `development`

```
terraform workspace new development
Created and switched to workspace "development"!
You're now on a new, empty workspace. Workspaces isolate their state, so
if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

You can validate that we are no longer in the `default` workspace by issuing a `terraform workspace show` command

```
terraform workspace show
development
```

You can also see that the state is empty for this workspace by issuing a `terraform show`

```
terraform show
No state.
```

**Task 3: Deploy Infrastructure within the Terraform development workspace**

Modify your `main.tf` to change the `region` of the `aws provider` block to `us-west-2`.

`main.tf`





```
# Configure the AWS Provider
provider "aws" {
  region = "us-west-2"
  default_tags {
    tags = {
      Owner      = "Acme"
      Provisioned = "Terraform"
    }
  }
}
```

Save your file and issue a `terraform plan` to see Terraform's dry run for execution

```
terraform plan
Plan: 26 to add, 0 to change, 0 to destroy.
```

Because the state information is new for the `development` workspace, Terraform will go and deploy all of the infrastructure declared in the configuration now into `us-west-2` which is our development region.

```
terraform apply

Do you want to perform these actions in workspace "development"?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
```

#### Task 4: Changing between Workspaces

To move between terraform workspaces you can use the `terraform workspace` command.

To move back to the infrastructure residing in `us-east-1` issue:

```
terraform workspace select default
```

To move back to the infrastructure residing in `us-west-2` issue:

```
terraform workspace select development
```

You can issue a `terraform show` in either workspace to see the resources that each of the workspaces is managing.





## Task 5: Utilizing the \${terraform.workspace} interpolation sequence within your configuration

Now that we see the benefit of isolating our resource state information using terraform workspaces, we may wish to reflect the workspace name information into our code base.

Within your Terraform configuration, you may include the name of the current workspace using the \${terraform.workspace} interpolation sequence.

Modify the environment default tag for `main.tf` inside the AWS provider to reflect the terraform workspace name

```
provider "aws" {
  region = "us-west-2"
  default_tags {
    tags = {
      Environment = terraform.workspace
      Owner       = "TF Hands On Lab"
      Project     = "Infrastructure as Code"
      Terraform   = "true"
    }
  }
}
```

```
# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
  id                               = "vpc-00d26110d48784808"
  ~ tags                           = {}
  ~ "Environment" = "demo_environment" -> "development"
  # (2 unchanged elements hidden)
}
~ tags_all                         = {}
~ "Environment" = "demo_environment" -> "development"
# (2 unchanged elements hidden)
}
# (14 unchanged attributes hidden)
}
```

## Reference

Terraform Workflow





## Lab: Terraform State Command

The `terraform state` command is used for advanced state management. As your Terraform usage becomes more advanced, there are some cases where you may need to modify the Terraform state. Rather than modify the state directly, the `terraform state` commands can be used in many cases instead.

- Task 1: Deploy Infrastructure Using Terraform
- Task 2: Utilize the `terraform show` command to show state information
- Task 3: Utilize the `terraform state` command to show state information
- Task 4: Utilize the `terraform state` command to list resource information
- Task 5: Utilize the `terraform state` command to show resource information

### Task 1: Deploy Infrastructure Using Terraform

If you have not already deployed your infrastructure from a previous lab create the following Terraform configuration files.

`terraform.tf`

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.0.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "3.1.0"
    }
  }
}
```





}

## main.tf

```

/*
Name: IaC Buildout for Terraform Associate Exam
Description: AWS Infrastructure Buildout
Contributors: Bryan and Gabe
*/

provider "aws" {
  region = "us-east-1"
  default_tags {
    tags = {
      Environment = terraform.workspace
      Owner       = "TF Hands On Lab"
      Project     = "Infrastructure as Code"
      Terraform   = "true"
    }
  }
}

#Retrieve the list of AZs in the current AWS region
data "aws_availability_zones" "available" {}
data "aws_region" "current" {}

#Define the VPC
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name        = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
}

#Deploy the private subnets
resource "aws_subnet" "private_subnets" {
  for_each           = var.private_subnets
  vpc_id             = aws_vpc.vpc.id
  cidr_block         = cidrsubnet(var.vpc_cidr, 8, each.value)
  availability_zone = tolist(data.aws_availability_zones.available.names)[
    each.value]

  tags = {
    Name        = each.key
    Terraform   = "true"
  }
}

```





```

    }
}

#Deploy the public subnets
resource "aws_subnet" "public_subnets" {
  for_each                    = var.public_subnets
  vpc_id                      = aws_vpc.vpc.id
  cidr_block                  = cidrsubnet(var.vpc_cidr, 8, each.value + 100)
  availability_zone           = tolist(data.aws_availability_zones.available.
    names)[each.value]
  map_public_ip_on_launch     = true

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}

#Create route tables for public and private subnets
resource "aws_route_table" "public_route_table" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.internet_gateway.id
    #nat_gateway_id = aws_nat_gateway.nat_gateway.id
  }
  tags = {
    Name      = "demo_public_rtb"
    Terraform = "true"
  }
}

resource "aws_route_table" "private_route_table" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    # gateway_id      = aws_internet_gateway.internet_gateway.id
    nat_gateway_id = aws_nat_gateway.nat_gateway.id
  }
  tags = {
    Name      = "demo_private_rtb"
    Terraform = "true"
  }
}

#Create route table associations
resource "aws_route_table_association" "public" {

```





```

depends_on      = [aws_subnet.public_subnets]
route_table_id = aws_route_table.public_route_table.id
for_each        = aws_subnet.public_subnets
subnet_id       = each.value.id
}

resource "aws_route_table_association" "private" {
depends_on      = [aws_subnet.private_subnets]
route_table_id = aws_route_table.private_route_table.id
for_each        = aws_subnet.private_subnets
subnet_id       = each.value.id
}

#Create Internet Gateway
resource "aws_internet_gateway" "internet_gateway" {
vpc_id = aws_vpc.vpc.id
tags = {
  Name = "demo_igw"
}
}

#Create EIP for NAT Gateway
resource "aws_eip" "nat_gateway_eip" {
vpc        = true
depends_on = [aws_internet_gateway.internet_gateway]
tags = {
  Name = "demo_igw_eip"
}
}

#Create NAT Gateway
resource "aws_nat_gateway" "nat_gateway" {
depends_on      = [aws_subnet.public_subnets]
allocation_id = aws_eip.nat_gateway_eip.id
subnet_id      = aws_subnet.public_subnets["public_subnet_1"].id
tags = {
  Name = "demo_nat_gateway"
}
}

# Terraform Data Block - To Lookup Latest Ubuntu 20.04 AMI Image
data "aws_ami" "ubuntu" {
most_recent = true

filter {
  name   = "name"
  values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
}
}

```





```

filter {
  name    = "virtualization-type"
  values  = ["hvm"]
}

owners = ["099720109477"]

}

# Terraform Resource Block - To Build EC2 instance in Public Subnet
resource "aws_instance" "web_server" {
  ami          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.
    ingress-ssh.id, aws_security_group.vpc-web.id]
  key_name      = aws_key_pair.generated.key_name
  connection {
    user          = "ubuntu"
    private_key   = tls_private_key.generated.private_key_pem
    host          = self.public_ip
  }
  associate_public_ip_address = true
  tags = {
    Name = "Web EC2 Server"
  }

  provisioner "local-exec" {
    command = "chmod 600 ${local_file.private_key_pem.filename}"
  }

  provisioner "remote-exec" {
    inline = [
      "git clone https://github.com/hashicorp/demo-terraform-101",
      "cp -a demo-terraform-101/. /tmp/",
      "sudo sh /tmp/assets/setup-web.sh",
    ]
  }
}

# Terraform Resource Block - Security Group to Allow Ping Traffic
resource "aws_security_group" "vpc-ping" {
  name          = "vpc-ping"
  vpc_id        = aws_vpc.vpc.id
  description   = "ICMP for Ping Access"
  ingress {
    description = "Allow ICMP Traffic"
    from_port   = -1
    to_port     = -1
    protocol    = "icmp"
  }
}

```





```

    cidr_blocks = ["0.0.0.0/0"]
}
egress {
  description = "Allow all ip and ports outbound"
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}
}

# Terraform Data Block - Lookup Ubuntu 16.04
data "aws_ami" "ubuntu_16_04" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }

  owners = ["099720109477"]
}

data "aws_ami" "windows_2019" {
  most_recent = true
  filter {
    name   = "name"
    values = ["Windows_Server-2019-English-Full-Base-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["801119661308"] # Canonical
}

resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content = tls_private_key.generated.private_key_pem
  filename = "MyAWSKey.pem"
}

resource "aws_key_pair" "generated" {
  key_name   = "MyAWSKey"
  public_key = tls_private_key.generated.public_key_openssh
}

```





```

lifecycle {
  ignore_changes = [key_name]
}
}

# Security Groups

resource "aws_security_group" "ingress-ssh" {
  name      = "allow-all-ssh"
  vpc_id    = aws_vpc.vpc.id
  ingress {
    cidr_blocks = [
      "0.0.0.0/0"
    ]
    from_port   = 22
    to_port    = 22
    protocol   = "tcp"
  }
  // Terraform removes the default rule
  egress {
    from_port   = 0
    to_port    = 0
    protocol   = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# Create Security Group - Web Traffic
resource "aws_security_group" "vpc-web" {
  name      = "vpc-web-${terraform.workspace}"
  vpc_id    = aws_vpc.vpc.id
  description = "Web Traffic"
  ingress {
    description = "Allow Port 80"
    from_port   = 80
    to_port    = 80
    protocol   = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "Allow Port 443"
    from_port   = 443
    to_port    = 443
    protocol   = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {

```





```

    description = "Allow all ip and ports outbound"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

```

`variables.tf`

```

variable "aws_region" {
  type    = string
  default = "us-east-1"
}

variable "vpc_name" {
  type    = string
  default = "demo_vpc"
}

variable "vpc_cidr" {
  type    = string
  default = "10.0.0.0/16"
}

variable "private_subnets" {
  default = {
    "private_subnet_1" = 1
    "private_subnet_2" = 2
    "private_subnet_3" = 3
  }
}

variable "public_subnets" {
  default = {
    "public_subnet_1" = 1
    "public_subnet_2" = 2
    "public_subnet_3" = 3
  }
}

```

Save your files and issue a `terraform init` and `terraform apply` to build out the infrastructure.

```

terraform init
terraform apply

```

```

Plan: 25 to add, 0 to change, 0 to destroy.

```





Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value: yes

## Task 2: Utilize the terraform show command to show state information

You can issue a `terraform show` to see the resources that were created by Terraform.

`terraform show`

```
# aws_eip.nat_gateway_eip:
resource "aws_eip" "nat_gateway_eip" {
    domain          = "vpc"
    id              = "eipalloc-0260c99a3a7a12677"
    network_border_group = "us-east-1"
    public_dns      = "ec2-3-92-117-57.compute-1.amazonaws.com"
    public_ip        = "3.92.117.57"
    public_ipv4_pool = "amazon"

...Redacted for brevity...

# tls_private_key.generated:
resource "tls_private_key" "generated" {
    algorithm          = "RSA"
    ecdsa_curve        = "P224"
    id                 = "999595a596d4b949afefcc27746a9c32b3582667"
    "
    private_key_pem     = (sensitive value)
    public_key_fingerprint_md5 = "8e:c1:c9:5c:05:85:62:b4:bb:fc:2e:45:5a:
                                b9:93:4a"
    public_key_openssh   = <<-EOT
        ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDLwvogPirs6RgzNkh68RK+Nq//Pqi1fucmSQfvz2N71ofzxal4KiaNabCzWqVfUVQZGUoJ2Ype8Lvv0383QUlgBZxZQjIMbBcAsr5le+jEkFY1tPQdhN43tLrlRsMhaKtbM3aiXMaNJDGNlEIvNsU948+yn4R0tnUmvpvmKms147RAToEm/2Yw2VK+Fi3C/BXLLBv7A0+1i+ZMtxCRQ1GCTqvcl12TsWxctPcR9LzzxeDBH8AnAjfAXf1E4eLvqALuE7Ap/yRv0g+zn60d1y7DtN25E9rC60ovV8EFWgjuWBVP64JSDw3v60BGsn6FU3
    EOT
    public_key_pem       = <<-EOT
        -----BEGIN PUBLIC KEY-----
        MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBCgKCAQEAY8L6ID4q70kYMzZIevESvjav/z6otX7nJkkH879je9aH82sZeComjWmws1qlX1FUGRlKCdmKXvC779N/N0FJYAWcWUIyDGwXALK+ZeratVRmdBDetrh4vXWHEtVH/oxJBWNbT0HYTeN7S65UbDIWirWzN2olzGjSXRjZRCLzbLvePPsp+ETrZ1Jr6b5iprNe00QE6BJv9mMNlSvhYtwv
```





```
wVy5Qb+wNPtYvmTLCXEUNRgk6r3pddk7FsXLT3EfS888XgwR/AJwI3wF39ROHi76
gC7h0wKf8kb9IPs5+tHdcuw7TduRPawutKL1fBBVoI7lgVT+uCug8N7+tARrJ+hV
NwIDAQAB
-----END PUBLIC KEY-----
EOT
rsa_bits
}
= 2048
```

You will note that there is a lot of information Terraform stores about all the resources it manages within its state file. To view and manage these resources in an easier way we will use the `terraform state` command.

### Task 3: Utilize the `terraform state` command to show state information

You can issue a `terraform state` to see the options for performing more granular operations related to the resources stored within Terraform state.

```
terraform state
```

Usage: `terraform [global options] state <subcommand> [options] [args]`

This **command** has subcommands **for** advanced state management.

These subcommands can be used to slice and dice the Terraform state. This is sometimes necessary **in** advanced cases. For your safety, all state management commands that modify the state create a timestamped backup of the state prior to making modifications.

The structure and output of the commands is specifically tailored to work well with the common Unix utilities such as `grep`, `awk`, etc. We recommend using those tools to perform more advanced state tasks.

Subcommands:

<code>list</code>	List resources <b>in</b> the state
<code>mv</code>	Move an item <b>in</b> the state
<code>pull</code>	Pull current state and output to stdout
<code>push</code>	Update remote state from a <b>local</b> state file
<code>replace-provider</code>	Replace provider <b>in</b> the state
<code>rm</code>	Remove instances from the state
<code>show</code>	Show a resource <b>in</b> the state



**Task 4: Utilize the terraform state command to list resource information**

You can issue a `terraform state list` in either workspace to see the resources that each of the workspaces is managing.

```
terraform state list
```

```
data.aws_ami.ubuntu
data.aws_ami.ubuntu_16_04
data.aws_ami.windows_2019
data.aws_availability_zones.available
data.aws_region.current
aws_eip.nat_gateway_eip
aws_instance.web_server
aws_internet_gateway.internet_gateway
aws_key_pair.generated
aws_nat_gateway.nat_gateway
aws_route_table.private_route_table
aws_route_table.public_route_table
aws_route_table_association.private["private_subnet_1"]
aws_route_table_association.private["private_subnet_2"]
aws_route_table_association.private["private_subnet_3"]
aws_route_table_association.public["public_subnet_1"]
aws_route_table_association.public["public_subnet_2"]
aws_route_table_association.public["public_subnet_3"]
aws_security_group.ingress-ssh
aws_security_group.vpc-ping
aws_security_group.vpc-web
aws_subnet.private_subnets["private_subnet_1"]
aws_subnet.private_subnets["private_subnet_2"]
aws_subnet.private_subnets["private_subnet_3"]
aws_subnet.public_subnets["public_subnet_1"]
aws_subnet.public_subnets["public_subnet_2"]
aws_subnet.public_subnets["public_subnet_3"]
aws_vpc.vpc
local_file.private_key_pem
tls_private_key.generated
```

**Task 5: Utilize the terraform state command to show resource information**

You can issue a `terraform state show` in either workspace to see the resources that each of the workspaces is managing.

```
terraform state show aws_instance.web_server
```





## Hands-On Labs

```
# aws_instance.web_server:
resource "aws_instance" "web_server" {
    ami                               = "ami-083654bd07b5da81d"
    arn                               = "arn:aws:ec2:us-east-1:1508140242758:instance/i-0f87913a4b4da9db5"
    associate_public_ip_address       = true
    availability_zone                 = "us-east-1b"
    cpu_core_count                   = 1
    cpu_threads_per_core             = 1
    disable_api_termination          = false
    ebs_optimized                    = false
    get_password_data                = false
    hibernation                      = false
    id                               = "i-0f87913a4b4da9db5"
    instance_initiated_shutdown_behavior = "stop"
    instance_state                   = "running"
    instance_type                    = "t2.micro"
    ipv6_address_count               = 0
    ipv6_addresses                   = []
    key_name                          = "MyAWSKey"
    monitoring                        = false
    primary_network_interface_id     = "eni-04039cf0944805906"
    private_dns                       = "ip-10-0-101-91.ec2.internal"
    private_ip                        = "10.0.101.91"
    public_ip                         = "3.236.193.131"
    secondary_private_ips             = []
    security_groups                  = [
        "sg-003059856f83334c0",
        "sg-0986589e4d7cdc719",
        "sg-0b9c402b6331dcbe",
    ]
    source_dest_check                = true
    subnet_id                         = "subnet-0ea34d249acab1fdb"
    tags                             = {
        "Name" = "Web EC2 Server"
    }
    tags_all                          = {
        "Environment" = "default"
        "Name"       = "Web EC2 Server"
        "Owner"      = "TF Hands On Lab"
        "Project"    = "Infrastructure as Code"
        "Terraform"   = "true"
    }
    tenancy                           = "default"
    vpc_security_group_ids            = [
        "sg-003059856f83334c0",
        "sg-0986589e4d7cdc719",
        "sg-0b9c402b6331dcbe",
    ]
}
```





```

capacity_reservation_specification {
    capacity_reservation_preference = "open"
}

credit_specification {
    cpu_credits = "standard"
}

enclave_options {
    enabled = false
}

metadata_options {
    http_endpoint           = "enabled"
    http_put_response_hop_limit = 1
    http_tokens             = "optional"
}

root_block_device {
    delete_on_termination = true
    device_name          = "/dev/sda1"
    encrypted             = false
    iops                  = 100
    tags                  = {}
    throughput             = 0
    volume_id              = "vol-04c74f0003ac59c4c"
    volume_size            = 8
    volume_type            = "gp2"
}
}

```

The `terraform state` command can also be used to manipulate state information. Rather than modify the state directly, the `terraform state` commands can be used in many cases instead. This includes moving, replacing, and overwriting resources and state information.

Note: Modifying state information directly is typically not required and should only be used in advanced cases. For your safety, all state management commands that modify the state create a timestamped backup of the state prior to making modifications. Exercise caution.

The output and command-line structure of the state subcommands is designed to be usable with Unix command-line tools such as grep, awk, and similar PowerShell commands.





## Reference

Terraform State Command





## Lab: Debugging Terraform

- Task 1: Enable Logging
- Task 2: Set Logging Path
- Task 3: Disable Logging

### Task 1: Enable Logging

Terraform has detailed logs which can be enabled by setting the TF\_LOG environment variable to any value. This will cause detailed logs to appear on stderr.

You can set TF\_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs, with TRACE being the most verbose.

Linux

```
export TF_LOG=TRACE
```

PowerShell

```
$env:TF_LOG="TRACE"
```

Run Terraform Apply.

```
terraform apply
```

Example Output

```
2020/03/20 13:14:41 [INFO] backend/local: apply calling Apply
2020/03/20 13:14:41 [INFO] terraform: building graph: GraphTypeApply
2020/03/20 13:14:41 [TRACE] Executing graph transform *terraform.
    ConfigTransformer
2020/03/20 13:14:41 [TRACE] ConfigTransformer: Starting for path:
2020/03/20 13:14:41 [TRACE] Completed graph transform *terraform.
    ConfigTransformer with new graph:
    data.vsphere_compute_cluster.cluster (prepare state) - *terraform.
        NodeApplyableResource
    data.vsphere_datacenter.dc (prepare state) - *terraform.
        NodeApplyableResource
    data.vsphere_datastore.datastore (prepare state) - *terraform.
        NodeApplyableResource
    data.vsphere_network.network (prepare state) - *terraform.
        NodeApplyableResource
    data.vsphere_virtual_machine.windows_template (prepare state) - *
        terraform.NodeApplyableResource
```





```

vsphere_tag.tag_release (prepare state) - *terraform.
  NodeApplyableResource
vsphere_tag.tag_tier (prepare state) - *terraform.NodeApplyableResource
vsphere_tag_category.category (prepare state) - *terraform.
  NodeApplyableResource
vsphere_virtual_machine.windows_vm (prepare state) - *terraform.
  NodeApplyableResource
-----
2020/03/20 13:14:41 [TRACE] Executing graph transform *terraform.
  DiffTransformer

```

## Task 2: Enable Logging Path

To persist logged output you can set TF\_LOG\_PATH in order to force the log to always be appended to a specific file when logging is enabled. Note that even when TF\_LOG\_PATH is set, TF\_LOG must be set in order for any logging to be enabled.

```
export TF_LOG_PATH="terraform_log.txt"
```

PowerShell

```
$env:TF_LOG_PATH="terraform_log.txt"
```

Run terraform init to see the initialization debugging information.

```
terraform init -upgrade
```

Open the **terraform\_log.txt** to see the contents of the debug trace for your terraform init. Experiment with removing the provider stanza within your Terraform configuration and run a **terraform plan** to debug how Terraform searches for where a provider is located.

## Task 3: Disable Logging

Terraform logging can be disabled by clearing the appropriate environment variable.

Linux

```
export TF_LOG=""
```

PowerShell

```
$env:TF_LOG=""
```





## Terraform Modules

Terraform configuration can be separated out into modules to better organize your configuration. This makes your code easier to read and reusable across your organization. A Terraform module is very simple: any set of Terraform configuration files in a folder is a module. Modules are the key ingredient to writing reusable and maintainable Terraform code. Complex configurations, team projects, and multi-repository codebases will benefit from modules. Get into the habit of using them wherever it makes sense.

- Task 1: Create a local Terraform module
- Task 2: Reference a module within Terraform Configuration
- Task 3: Terraform module reuse

### Task 1: Create a local Terraform Module

A Terraform module is just a set of Terraform configuration files. Modules are just Terraform configurations inside a folder - there's nothing special about them. In fact, the code you've been writing so far is a module: the root module. For this lab, we'll create a local module for a new server configuration.

#### Step 1.1

Create a new directory called `server` in your `/workspace/terraform` directory and create a new file inside of it called `server.tf`.

#### Step 1.2

Edit the file `server/server.tf`, with the following contents:

```
variable "ami" {}
variable "size" {
  default = "t2.micro"
}
variable "subnet_id" {}
variable "security_groups" {
  type = list(any)
}
resource "aws_instance" "web" {
  ami           = var.ami
  instance_type = var.size
```





```

subnet_id          = var.subnet_id
vpc_security_group_ids = var.security_groups

tags = {
  "Name"      = "Server from Module"
  "Environment" = "Training"
}
}

output "public_ip" {
  value = aws_instance.web.public_ip
}

output "public_dns" {
  value = aws_instance.web.public_dns
}

```

### Step 1.3

In your root configuration (also called your root module) /workspace/terraform/main.tf, we can call our new `server` module with a Terraform module block. Remember that terraform only works with the configuration files that are in it's current working directory. Modules allow us to reference Terraform configuration that lives outside of our working directory. In this case we will incorporate all configuration that is both inside our working directory (root module) and inside the `server` directory (child module).

```

module "server" {
  source      = "./server"
  ami         = data.aws_ami.ubuntu.id
  subnet_id   = aws_subnet.public_subnets["public_subnet_3"].id
  security_groups = [
    aws_security_group.vpc-ping.id,
    aws_security_group.ingress-ssh.id,
    aws_security_group.vpc-web.id
  ]
}

```

### Step 1.4 Install and Apply Module

Now run `terraform init` to install the module. Terraform configuration files located within modules are pulled down by Terraform during initialization, so any time you add or update a module version you must run a `terraform init`.





```
terraform init
```

You can see that our configuration now depends on this module to be installed and used by using the `terraform providers` command.

```
terraform providers
```

Providers required by configuration:

```
-- provider[registry.terraform.io/hashicorp/aws] ~> 3.0
-- provider[registry.terraform.io/hashicorp/http] 2.1.0
-- provider[registry.terraform.io/hashicorp/random] 3.1.0
-- provider[registry.terraform.io/hashicorp/local] 2.1.0
-- provider[registry.terraform.io/hashicorp/tls] 3.1.0
|-- module.server
  |-- provider[registry.terraform.io/hashicorp/aws]
```

Providers required by state:

```
provider[registry.terraform.io/hashicorp/aws]
provider[registry.terraform.io/hashicorp/local]
provider[registry.terraform.io/hashicorp/random]
provider[registry.terraform.io/hashicorp/tls]
```

Run `terraform apply` to create a new server using the `server` module. It may take a few minutes for the server to be built using the module.

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
module.server.aws_instance.web: Creating...
module.server.aws_instance.web: Still creating... [10s elapsed]
```

## Task 2: Reference a Module within Terraform Configuration

When used, Terraform modules will be listed within Terraform's state file and can be referenced using their module name.

```
terraform state list
```





```
# Resource defined in root module/working directory
aws_instance.web_server

# Resource defined in `server` module
module.server.aws_instance.web
```

We can look at all the details of the server created using our `server` module.

```
terraform show module.server.aws_instance.web
```

We can add two output blocks to our `main.tf` to report back the IP and DNS information from our `server` module. Notice how Terraform references (interpolation syntax) information about the server build from a module.

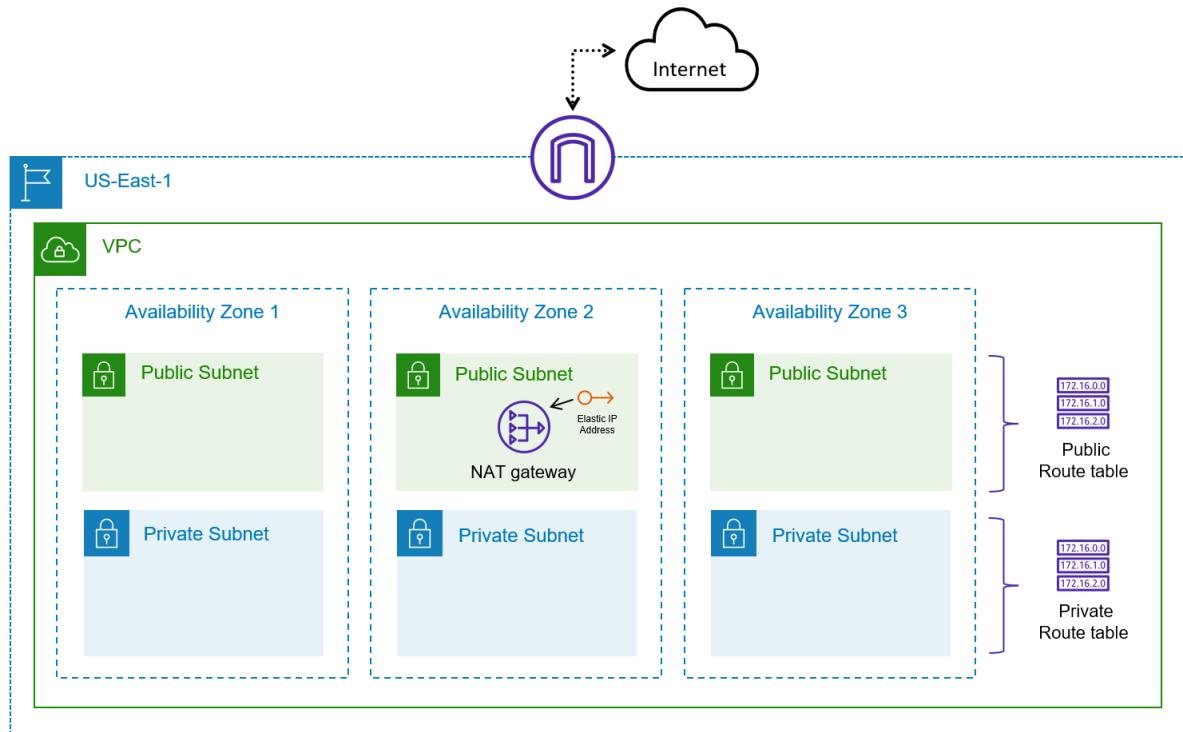
```
output "public_ip" {
  value = module.server.public_ip
}

output "public_dns" {
  value = module.server.public_dns
}
```

### Task 3: Reuse the module to build a server in a different subnet

One of the benefits of Terraform modules is that they can easily be reused across your organization. Let's use our local module again to build out another server in a separate subnet.



**Figure 1:** Desired Infrastructure

```
module "server_subnet_1" {
  source          = "./server"
  ami             = data.aws_ami.ubuntu.id
  subnet_id       = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [
    aws_security_group.vpc-ping.id,
    aws_security_group.ingress-ssh.id,
    aws_security_group.vpc-web.id
  ]
}
```

Run `terraform apply` to create a new server using the `server` module. It may take a few minutes for the server to be built using the module.

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
module.server_subnet_1.aws_instance.web: Creating...
```





Hands-On Labs

```
module.server_subnet_1.aws_instance.web: Still creating... [10s elapsed]
```

You

```
output "public_ip_server_subnet_1" {  
    value = module.server_subnet_1.public_ip  
}  
  
output "public_dns_server_subnet_1" {  
    value = module.server_subnet_1.public_dns  
}
```





## Terraform Module Sources

Modules can be sourced from a number of different locations, including both local and remote sources. The Terraform Module Registry, HTTP urls and S3 buckets are examples of remote sources, while folders and subfolders are examples of local sources. Support for various module sources allow you to include Terraform configuration from a variety of locations while still providing proper organization of code.

- Task 1: Source a local Terraform module
- Task 2: Explore the Public Module Registry and install a module
- Task 3: Source a module from GitHub

### Task 1: Source a local Terraform module

While not required, local modules are commonly saved in a folder named `modules`, and each module is named for its respective function inside that folder. In support of this practice let's move our `server` module into a `modules` directory and observe the impact.

#### Step 1.1 - Refactor to use local modules directory

Create a `modules` directory and move your `server` directory into this directory as a subdirectory.

```
|-- modules
|   |-- server
|       |-- server.tf
```

Update the `source` of your module block to point to the modified source.

```
module "server" {
  source      = "./modules/server"
  ami         = data.aws_ami.ubuntu.id
  subnet_id   = aws_subnet.public_subnets["public_subnet_3"].id
  security_groups = [aws_security_group.vpc-ping.id,
                     aws_security_group.ingress-ssh.id,
                     aws_security_group.vpc-web.id]
}

module "server_subnet_1" {
  source      = "./modules/server"
  ami         = data.aws_ami.ubuntu.id
  subnet_id   = aws_subnet.public_subnets["public_subnet_1"].id
```





```

    security_groups = [aws_security_group.vpc-ping.id,
    aws_security_group.ingress-ssh.id,
    aws_security_group.vpc-web.id]
}

```

Any time that the source of a module is update, the working directory (root module) needs to be reinitialized.

```
terraform init
```

We can follow that by issuing a `terraform plan` to make sure that moving our module source did not incur any changes to our infrastructure. This is expected because we have not changed any of the Terraform configuration, just the source of our module that was moved.

```
terraform plan
```

```
No changes. Your infrastructure matches the configuration.
```

```
Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are
needed.
```

### Step 1.2 - Create a new module source locally

Within our `modules` directory let's create another directory where we will include some enhancements to our `server` module. We will create these enhancments in a new directory called `web_server` and make a copy of our `server.tf` folder to this directory.

```

|-- modules
|   |-- server
|   |   |-- server.tf
|   |-- web_server
|   |   |-- server.tf

```

Let's update the `server.tf` inside our our `web_server` directory to now to allow us to provision our web application.

```

variable "ami" {}
variable "size" {
  default = "t2.micro"
}
variable "subnet_id" {}

variable "user" {}

```





```
variable "security_groups" {
  type = list(any)
}

variable "key_name" {

}

variable "private_key" {

}

resource "aws_instance" "web" {
  ami                  = var.ami
  instance_type        = var.size
  subnet_id            = var.subnet_id
  vpc_security_group_ids = var.security_groups
  associate_public_ip_address = true
  key_name             = var.key_name
  connection {
    user      = var.user
    private_key = var.private_key
    host       = self.public_ip
  }
}

provisioner "remote-exec" {
  inline = [
    "sudo rm -rf /tmp",
    "sudo git clone https://github.com/hashicorp/demo-terraform-101 /tmp",
    "",
    "sudo sh /tmp/assets/setup-web.sh",
  ]
}

tags = {
  "Name"      = "Web Server from Module"
  "Environment" = "Training"
}

output "public_ip" {
  value = aws_instance.web.public_ip
}

output "public_dns" {
  value = aws_instance.web.public_dns
}
```





```
}
```

### Step 1.2 - Update our server\_subnet\_1 module to use the new web\_server source

Let's now update our `server_subnet_1` module to use the new `web_server` source configuration by updating the `source` argument.

```
module "server_subnet_1" {
  source      = "./modules/web_server"
  ami         = data.aws_ami.ubuntu.id
  key_name    = aws_key_pair.generated.key_name
  user        = "ubuntu"
  private_key = tls_private_key.generated.private_key_pem
  subnet_id   = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id,
                     aws_security_group.ingress-ssh.id,
                     aws_security_group.vpc-web.id]
}
```

Any time that the source of a module is updated or added to our configuration, the working directory needs to be reinitialized.

```
terraform init
```

You can make sure that the syntax is validate and formated properly by running the following:

```
terraform validate
terraform fmt -recursive
```

### Step 1.2 - Apply our configuration with the updated module

Now that we have updated the source of our `server_subnet_1` module, we are ready to use it. Let's first see the ramifications of this change via a `terraform plan`

```
terraform plan
```

You will see that because we are now updating our server configuration via the new module to specify a key pair, that the server will need to be rebuilt.

```
Plan: 1 to add, 0 to change, 1 to destroy.
```

```
terraform apply
```





Now we will see that our new `web_server` module deploys a web application when used, versus the `server` module which simply deploys a generic server. We can of course switch between which modules are used by updating the `source` argument of our module - just be sure to run a `terraform plan` after making any changes to fully understand the impact of that change.

## Task 2: Explore the Public Module Registry and install a module

Terraform Public Registry is an index of modules shared publicly. This public registry is the easiest way to get started with Terraform and find modules created by others in the community. You can also use a private registry as a feature of Terraform Cloud/Terraform Enterprise.

Modules on the public Terraform Registry can be sourced using a registry source address of the form `//`, with each module's information page on the registry site including the exact address to use.

We will use the AWS Autoscaling module to deploy an AWS Autoscaling group to our environment. Update your `main.tf` to include this module from the Terraform Module Registry.

```
module "autoscaling" {
  source  = "terraform-aws-modules/autoscaling/aws"
  version = "4.9.0"

  # Autoscaling group
  name = "myasg"

  vpc_zone_identifier = [aws_subnet.private_subnets["private_subnet_1"].id
    ,
    aws_subnet.private_subnets["private_subnet_2"].id,
    aws_subnet.private_subnets["private_subnet_3"].id]
  min_size          = 0
  max_size          = 1
  desired_capacity = 1

  # Launch template
  use_lt            = true
  create_lt         = true

  image_id          = data.aws_ami.ubuntu.id
  instance_type     = "t3.micro"

  tags_as_map = {
    Name = "Web EC2 Server 2"
  }
}
```





Any time that the source of a module is updated or added to our configuration, the working directory needs to be reinitialized.

```
terraform init
```

We can follow that by issuing a `terraform plan` to see that additional resources that will be added by using the module.

```
terraform plan
```

Once we are happy with how the module is behaving we can issue a `terraform apply`

```
terraform apply
```

### Task 3: Source a module from GitHub

Another source of modules that we can use are those that are published directly on GitHub. Terraform will recognize unprefixed `github.com` URLs and interpret them automatically as Git repository sources. Let's update the `source` of our autoscaling group module from the Public Module registry to use `github.com` instead. This will require us to remove the `version` argument from our module block.

```
module "autoscaling" {
  source = "github.com/terraform-aws-modules/terraform-aws-autoscaling?ref
            =v4.9.0"

  # Autoscaling group
  name = "myasg"

  vpc_zone_identifier = [aws_subnet.private_subnets["private_subnet_1"].id
    ,
    aws_subnet.private_subnets["private_subnet_2"].id,
    aws_subnet.private_subnets["private_subnet_3"].id]
  min_size          = 0
  max_size          = 1
  desired_capacity = 1

  # Launch template
  use_lt      = true
  create_lt   = true

  image_id      = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags_as_map = {
    Name = "Web EC2 Server 2"
}
```





```
    }  
}  
}
```

These GitHub schemes are treated as convenient aliases for the general Git repository address scheme, and so they obtain credentials in the same way and support the ref argument for selecting a specific revision. You will need to configure credentials in particular to access private repositories.

Now that our module source has been change we must reinitialize our working directory, plan and apply.

```
terraform init
```

We can follow that by issuing a `terraform plan` to see that additional resources that will be added by using the module.

```
terraform plan
```

Once we are happy with how the module is behaving we can issue a `terraform apply`

```
terraform apply
```





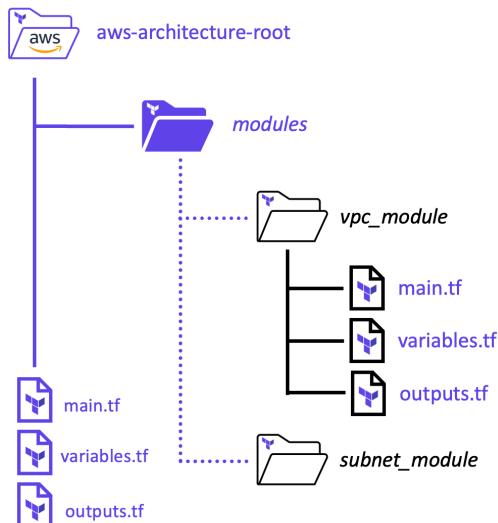
## Terraform Modules Inputs and Outputs

To make a Terraform module configurable you can add input parameters to the module. These are defined within the module using input variables. A module can also return values to the configuration that called the module. These module returns or outputs are defined using terraform output blocks.

- Task 1: Refactor module using code organization principles
- Task 2: Required and Optional Module inputs
- Task 3: Module outputs and returns
- Task 4: Reference module outputs

### Task 1: Refactor module using code organization principles

You can create a module with a single .tf file, or use any other file structure you like. Modules can be a bit daunting at first. We are suddenly working with a number of files in different file hierarchies, and those files are referencing each other in a few different ways. An example of this can be found in the diagram below:



The example modules contain the following files:

`main.tf` will contain the main set of configuration for your module. You can also create other configuration files and organize them however makes sense for your project.

`variables.tf` will contain the variable definitions for your module. When your module is used by others, the variables will be configured as arguments in the module block. Since all Terraform values





must be defined, any variables that are not given a default value will become required arguments. Variables with default values can also be provided as module arguments, overriding the default value.

`outputs.tf` will contain the output definitions for your module. Module outputs are made available to the configuration using the module, so they are often used to pass information about the parts of your infrastructure defined by the module to other parts of your configuration.

### Step 1.1 - Refactor the server module

Our `server` module is very simplistic, and we want to incorporate the code organization principles to follow the file structure above. Create three new files inside the `server` folder: `main.tf`, `variables.tf`, `outputs.tf`

Copy the appropriate code out of the `server.tf` and place it into these appropriate files:

`main.tf`

```
resource "aws_instance" "web" {
  ami           = var.ami
  instance_type = var.size
  subnet_id     = var.subnet_id
  vpc_security_group_ids = var.security_groups

  tags = {
    "Name"      = "Server from Module"
    "Environment" = "Training"
  }
}
```

`variables.tf`

```
variable "ami" {}
variable "size" {
  default = "t2.micro"
}
variable "subnet_id" {}
variable "security_groups" {
  type = list(any)
}
```

`outputs.tf`

```
output "public_ip" {
```





```

    value = aws_instance.web.public_ip
}

output "public_dns" {
    value      = aws_instance.web.public_dns
}

```

You can run a `terraform init` along with a `terraform plan` and there should be no changes if we refactored the module correctly.

```

terraform init
terraform plan

```

## Task 2: Module required and optional inputs

Variables within modules work almost exactly the same way that they do for the root module. When you run a Terraform command on your root configuration, there are various ways to set variable values, such as passing them on the commandline, or with a `.tfvars` file. When using a module, variables are set by passing arguments to the module in your configuration. You will set some of these variables when calling this module from your root module's `main.tf`.

Variables defined in modules that aren't given a default value are required, and so must be set whenever the module is used. Comment out the `size` variable within the `server` module to make it required.

`variables.tf`

```

variable "ami" {}
variable "size" {
    # default = "t2.micro"
}
variable "subnet_id" {}
variable "security_groups" {
    type = list(any)
}

```

Run a `terraform validate` from the root module/working directory and notice that the module now requires the `size` argument to be passed into the module block, because it is now required.

```

terraform validate

| Error: Missing required argument
|
|   on main.tf line 316, in module "server":
| 316: module "server" {

```





| The argument "size" is required, but no definition was found.

Update the `server` module block to know specify the new required argument: `size`

```
module "server" {
  source      = "./modules/server"
  ami         = data.aws_ami.ubuntu.id
  size        = "t2.micro"
  subnet_id   = aws_subnet.public_subnets["public_subnet_3"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.
    ingress-ssh.id, aws_security_group.vpc-web.id]
}
```

Validate that the all required inputs into the `server` module block have been satisfied.

```
terraform validate
Success! The configuration is valid.
```

### Task 3: Module outputs and returns

Like variables, outputs in modules perform the same function as they do in the root module but are accessed in a different way. A module's outputs can be accessed as read-only attributes on the module object, which is available within the configuration that calls the module. You can reference these outputs in expressions as `module.<MODULE NAME>.<OUTPUT NAME>`.

Outputs are required for common modules that define resources. Variables and outputs are used to infer dependencies between modules and resources. Without any outputs we cannot properly order your module in relation to their Terraform configurations.

We can add a return to our `server` module by adding an output block within the module. Add a `size` variable within the `outputs.tf` of our `server` module.

`output.tf`

```
output "public_ip" {
  description = "IP Address of server built with Server Module"
  value       = aws_instance.web.public_ip
}

output "public_dns" {
  description = "DNS Address of server built with Server Module"
  value       = aws_instance.web.public_dns
}
```





```
output "size" {
  description = "Size of server built with Server Module"
  value       = aws_instance.web.instance_type
}
```

Add an output inside the `main.tf` of our working directory to reference the new output provided by the module.

```
output "size" {
  value = module.server.size
}
```

```
terraform init
terraform apply
```

Outputs should have meaningful descriptions you should make an effort to output all the useful values root modules would want to reference or share with modules. Particularly for open source or heavily used modules, expose all outputs that have potential for consumption.

#### Task 4: Reference module outputs

In order to reference items that are returned by modules (by the module's `outputs.tf` file) you must use the interpolation syntax referring to the output name returned by the module. Eg: `module.server.public_ip`

`outputs.tf` of server module

```
output "public_ip" {
  description = "IP Address of server built with Server Module"
  value       = aws_instance.web.public_ip
}
```

Refering to the `public_ip` of the server module within the root module:

`main.tf` of root module:

```
output "public_ip" {
  value = module.server.public_ip
}
```

Notice that both output blocks have an output named `public_ip`, but their values reference a different interpolation. The output within the module itself references the value of the `public_ip` argument of the resource itself, while the output within the root module/working directory's configuration references the value of `public_ip` returned by the module.





## Terraform Module Scope

Deciding what infrastructure to include is the one of the most challenging aspects about creating a new Terraform module.

Modules should be opinionated and designed to do one thing well. If a module's function or purpose is hard to explain, the module is probably too complex. When initially scoping your module, aim for small and simple to start.

- Task 1: Resources within Child Modules
- Task 2: Scoping Module Inputs and Outputs
- Task 3: Reference Child Module Outputs
- Task 4: Invalid Module References

### Task 1: Resources within Child Modules

In principle any combination of resources and other constructs can be factored out into a module, but over-using modules can make your overall Terraform configuration harder to understand and maintain, so we recommend moderation. A good module should raise the level of abstraction by describing a new concept in your architecture that is constructed from resource types offered by providers.

Let's take a closer look at the auto scaling group module that we are calling to further understand which resources are used to construct the module. Inside our `main.tf` we can see that the autoscaling module we are calling is being sourced from the Terraform Public Module registry, and we are passing 10 inputs into the module.

`main.tf`

```
module "autoscaling" {
  source  = "terraform-aws-modules/autoscaling/aws"
  version = "4.9.0"

  # Autoscaling group
  name = "myasg"

  vpc_zone_identifier = [aws_subnet.private_subnets["private_subnet_1"].id,
                        aws_subnet.private_subnets["private_subnet_2"].id, aws_subnet.
                        private_subnets["private_subnet_3"].id]
  min_size           = 0
  max_size           = 1
  desired_capacity   = 1
```





```
# Launch template
use_lt     = true
create_lt = true

image_id      = data.aws_ami.ubuntu.id
instance_type = "t3.micro"

tags_as_map = {
  Name = "Web EC2 Server 2"
}

}
```

The screenshot shows the Terraform Registry interface. At the top, there's a navigation bar with the Terraform logo, 'Registry', a search bar ('Search Providers and Modules'), and links for 'Browse', 'Publish', and 'Sign-in'. Below the navigation, the main content area displays the 'aws autoscaling' module. It features the AWS logo, the module name 'autoscaling' with a blue 'AWS' badge, and a brief description: 'Terraform module which creates Auto Scaling resources on AWS'. To the right, there's a 'Provision Instructions' section with a code snippet for Terraform configuration:

```
module "autoscaling" {
  source  = "terraform-aws-modules/autoscaling"
  version = "4.9.0"
  # insert the 57 required variables here
}
```

At the bottom of the module page, there are links for 'Readme', 'Inputs (86)', 'Outputs (20)', 'Dependency (1)', and 'Resources (4)'.

**Figure 1:** Autoscaling Module

This module combines four different AWS resources, accepts up to 86 inputs and returns 20 possible outputs. It also has a dependency on the AWS provider.





## Resources

This is the list of resources that the module *may* create. The module can create zero or more of each of these resources depending on the `count` value. The count value is determined at runtime. The goal of this page is to present the types of resources that may be created.

This list contains all the resources this plus any submodules may create. When using this module, it may create less resources if you use a submodule.

This module defines 4 resources .

- `aws_autoscaling_group.this`
- `aws_autoscaling_schedule.this`
- `aws_launch_configuration.this`
- `aws_launch_template.this`

**Figure 2:** Autoscaling Module

```
aws_autoscaling_group
aws_autoscaling_schedule
aws_launch_configuration
aws_launch_template
```

Execute a `terraform init` and `terraform apply` to build out the infrastructure including the Auto Scaling group module. View the items that this module built by running a `terraform state list`

```
terraform state list

module.autoscaling.data.aws_default_tags.current
module.autoscaling.aws_autoscaling_group.this[0]
module.autoscaling.aws_launch_template.this[0]
```

This module built out an autoscaling group and launch template as well as looked up default tags via a data block. You will notice that while this module could have built out more resources, these items were all that needed based on the configuration we passed into the module.

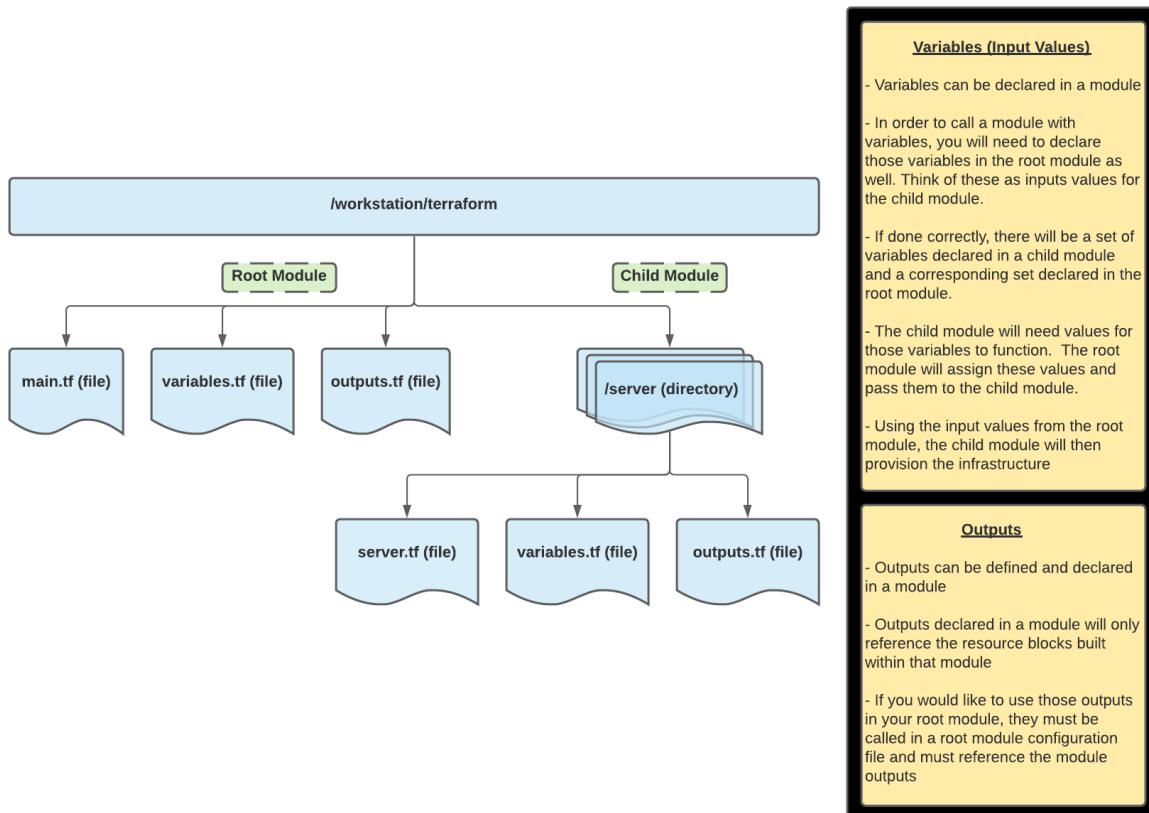
## Task 2: Scoping Module Inputs and Outputs

When looking at the scope of a module block, this diagram provides a helpful way to understand the relationship between root modules and child modules.





## Hands-On Labs



Modules are simply terraform configuration files that have optional/required inputs and will return a specified number of outputs. The configuration within the module can be akin to a black box, as the module is abstracting away the configuration blocks which it contains. The module code for child modules is still typically available for us to review if we like, but not required to be able to get resources built out without having to worry about all of the details within the child module.

When creating a module, consider which resource arguments to expose to module end users as input variables. In our example, the autoscaling module provides up to 86 different inputs that can be used for configuring the ASG. Variables declared in child module map to arguments within root module block, where outputs declared in child module map to attributes that can be used by the root module. You should also consider which values to add as outputs, since outputs are the only supported way for users to get information about resources configured by the module. In our example, the autoscaling module provides 20 available outputs that the module will return.

To view the outputs returned by our autoscaling module and their values you can issue the following:

```
terraform console
```





```
> module.autoscaling

{
  "autoscaling_group_arn" = "arn:aws:autoscaling:us-east-1:508140242758:
    autoScalingGroup:2a33891c-6c8b-4434-9386-bf1f1574523d:
    autoScalingGroupName/myasg-20211218123153166900000003"
  "autoscaling_group_availability_zones" = toset([
    "us-east-1b",
    "us-east-1c",
    "us-east-1d",
  ])
  "autoscaling_group_default_cooldown" = 300
  "autoscaling_group_desired_capacity" = 1
  "autoscaling_group_health_check_grace_period" = 300
  "autoscaling_group_health_check_type" = "EC2"
  "autoscaling_group_id" = "myasg-20211218123153166900000003"
  "autoscaling_group_load_balancers" = toset(null) /* of string */
  "autoscaling_group_max_size" = 1
  "autoscaling_group_min_size" = 0
  "autoscaling_group_name" = "myasg-20211218123153166900000003"
  "autoscaling_group_target_group_arns" = toset(null) /* of string */
  "autoscaling_group_vpc_zone_identifier" = toset([
    "subnet-01dcabb8c2474bd4f",
    "subnet-0d29fb83c81f6278b",
    "subnet-0e509d03fb7841876",
  ])
  "autoscaling_schedule_arns" = (known after apply)
  "launch_configuration_arn" = (known after apply)
  "launch_configuration_id" = (known after apply)
  "launch_configuration_name" = (known after apply)
  "launch_template_arn" = "arn:aws:ec2:us-east-1:508140242758:launch-
    template/lt-0bdc46cf90ccdad50"
  "launch_template_id" = "lt-0bdc46cf90ccdad50"
  "launch_template_latest_version" = 1
}
```

### Task 3: Reference Child Module Outputs

In order to reference items that are returned by modules (by the child module's `outputs.tf` file) you must use the interpolation syntax referring to the output name returned by the module. Eg: `module.autoscaling.autoscaling_group_max_size`

Referring to the `autoscaling_group_max_size` of the autoscaling module within the root module:

Let's add an output block inside the `main.tf` of root module to showcase the `autoscaling_group_max_size` returned by the autoscaling group child module.





```
output "asg_group_size" {
  value = module.autoscaling.autoscaling_group_max_size
}
```

```
terraform apply
```

```
Outputs:
asg_group_size = 1
```

#### Task 4: Invalid Module References

Module outputs are the only supported way for users to get information about resources configured within the child module. Individual resource arguments are not accessible outside the child module.

If we look at the `outputs.tf` of autoscaling group child module we can see the one of the items returned by this module is the `autoscaling_group_max_size`. Looking at the code this is returned based on the value of the `aws_autoscaling_group.this[0].max_size` resource. We don't need to be too occupied with how this value is set since the module is abstracting away these complexities for us.

```
output "autoscaling_group_max_size" {
  description = "The maximum size of the autoscale group"
  value       = try(aws_autoscaling_group.this[0].max_size, "")
```

All we need to be concerned about is the `autoscaling_group_max_size` output that is returned. In fact, unless a child module provides us with an output that we can consume we will not be able to reference items within the child module. Individual resource arguments are not accessible outside the child module.

```
module "my-module" {
  # Source can be any URL or file path
  source = "../../my-module"

  argument_1 = "value"
  argument_2 = "value"
}

# Valid
output "example" {
  value = module.my-module.public_ip
}
```





```
# Invalid
output "example" {
  value = module.my-module.aws_instance.db.public_ip
}
```

To showcase this, create an output block within the root module `main.tf` that references a resource argument directly within the autoscaling child module.

`main.tf`

```
output "asg_group_size" {
  value = module.autoscaling.aws_autoscaling_group.this[0].max_size
}
```

`terraform validate`

```
Error: Unsupported attribute

  on main.tf line 382, in output "asg_group_size":
382:   value = module.autoscaling.aws_autoscaling_group.this[0].max_size
                  |
                  |
                  |-----|
                  | module.autoscaling is a object, known only after apply
                  |
This object does not have an attribute named "aws_autoscaling_group".
```

Now correct this to reference the valid output returned by the module:

```
output "asg_group_size" {
  value = module.autoscaling.autoscaling_group_max_size
}
```

`terraform validate`

```
Success! The configuration is valid.
```

It is therefore a best practice when creating terraform modules to output as much information as possible, even if you do not currently have a use for it. This will make your module more useful for end users who will often use multiple modules, using outputs from one module as inputs for the next.

## Notes about building Terraform Modules

When building a module, consider three areas:

- **Encapsulation:** Group infrastructure that is always deployed together. Including more infrastructure in a module makes it easier for an end user to deploy that infrastructure but makes the





## Hands-On Labs

module's purpose and requirements harder to understand

- **Privileges:** Restrict modules to privilege boundaries. If infrastructure in the module is the responsibility of more than one group, using that module could accidentally violate segregation of duties. Only group resources within privilege boundaries to increase infrastructure segregation and secure your infrastructure
- **Volatility:** Separate long-lived infrastructure from short-lived. For example, database infrastructure is relatively static while teams could deploy application servers multiple times a day. Managing database infrastructure in the same module as application servers exposes infrastructure that stores state to unnecessary churn and risk.

A simple way to get start with creating modules is to:

- Always aim to deliver a module that works for at least 80% of use cases.
- Never code for edge cases in modules. An edge case is rare. A module should be a reusable block of code.
- A module should have a narrow scope and should not do multiple things.
- The module should only expose the most commonly modified arguments as variables. Initially, the module should only support variables that you are most likely to need.





## Terraform Modules - Public Module Registry

Hashicorp maintains a public registry that helps you to consume Terraform modules from others. The Terraform Public Registry is an index of modules shared publicly and is the easiest way to get started with Terraform and find modules created by others in the community. It includes support for module versioning and searchable and filterable list of available modules for quickly deploying common infrastructure configurations.

Modules on the public Terraform Registry can be sourced using a registry source address of the form <NAMESPACE>/<NAME>/<PROVIDER>, with each module's information page on the registry site including the exact address to use.

- Task 1: Consuming Modules from the Terraform Module Registry
- Task 2: Exploring other modules from the Terraform Module Registry
- Task 3: Publishing to the Terraform Public Module Registry

### Task 1: Consuming Modules from the Terraform Module Registry

In previous labs we began using modules from the Terraform Public Module Registry, including the auto scaling module. Let's add an S3 bucket to our configuration using the S3 public module.

main.tf

```
module "s3-bucket" {  
  source  = "terraform-aws-modules/s3-bucket/aws"  
  version = "2.11.1"  
}  
  
output "s3_bucket_name" {  
  value = module.s3-bucket.s3_bucket_bucket_domain_name  
}
```

```
terraform init  
terraform plan
```

```
# module.s3-bucket.aws_s3_bucket.this[0] will be created  
+ resource "aws_s3_bucket" "this" {  
    + acceleration_status      = (known after apply)  
    + acl                      = "private"  
    + arn                      = (known after apply)  
    + bucket                   = (known after apply)  
    + bucket_domain_name       = (known after apply)
```





```
+ bucketRegionalDomainName = (known after apply)
+ forceDestroy           = false
+ hostedZoneId          = (known after apply)
+ id                     = (known after apply)
+ region                = (known after apply)
+ requestPayer           = (known after apply)
+ tagsAll               = (known after apply)
+ websiteDomain          = (known after apply)
+ websiteEndpoint         = (known after apply)

+ versioning {
    + enabled      = (known after apply)
    + mfaDelete   = (known after apply)
}
}

# module.s3-bucket.aws_s3_bucket_public_access_block.this[0] will be
# created
+ resource "aws_s3_bucket_public_access_block" "this" {
    + blockPublicAcls     = false
    + blockPublicPolicy   = false
    + bucket              = (known after apply)
    + id                  = (known after apply)
    + ignorePublicAcls   = false
    + restrictPublicBuckets = false
}
```

`terraform apply`

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only '`yes`' will be accepted to approve.

Enter a value: yes

`s3_bucket_name = "terraform-20211220113638832700000001.s3.amazonaws.com"`

## Task 2: Exploring other modules from the Terraform Module Registry

Another useful module that we may wish to utilize is the VPC Module. This is a simple Terraform module for creating VPC resource in AWS

Let's this to our configuration inside the `main.tf` of our root module:

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
```





```

name = "my-vpc-terraform"
cidr = "10.0.0.0/16"

azs          = ["us-east-1a", "us-east-1b", "us-east-1c"]
private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

enable_nat_gateway = true
enable_vpn_gateway = true

tags = {
  Name = "VPC from Module"
  Terraform = "true"
  Environment = "dev"
}
}

```

```

terraform init
terraform plan
terraform apply

```

Subnets (6) <a href="#">Info</a>							
<a href="#">Actions</a> <a href="#">Create subnet</a>							
<input type="text"/> Filter subnets							
<input style="border: 1px solid #ccc; padding: 2px; margin-right: 10px;" type="button" value="Name: VPC from Module"/> <span style="border: 1px solid #ccc; padding: 2px 10px; border-radius: 5px;">Name: VPC from Module</span>		<a href="#">Clear filters</a>					
Name	Subnet ID	State	VPC	IPv4 CIDR			
<input type="checkbox"/> VPC from Module	subnet-075e03f26dfd85602	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.103.0/24			
<input type="checkbox"/> VPC from Module	subnet-0167f1c57ce996236	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.2.0/24			
<input type="checkbox"/> VPC from Module	subnet-088ecefa1935e9466	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.102.0/24			
<input type="checkbox"/> VPC from Module	subnet-0a8b4e77f589d672d	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.1.0/24			
<input type="checkbox"/> VPC from Module	subnet-0d5c802b63cc29328	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.101.0/24			
<input type="checkbox"/> VPC from Module	subnet-074d12cc14fb75957	<span style="color: green;">Available</span>	vpc-04756fbbeb80768e50   VP...	10.0.3.0/24			

**Figure 1:** VPC Module

You can see that we can simplify our code base through the use of modules. The last two examples showcased building out an entire AWS VPC with private and public subnets, route tables, NAT gateways, VPN and Internet Gateways along with an S3 Bucket all via Terraform modules.





### Task 3: Publishing to the Terraform Public Module Registry

Anyone can publish and share modules on the Terraform Registry, but there are some requirements that you should be aware of:

- The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.
- Module repositories must use a naming format: `terraform-<PROVIDER>-<NAME>` where reflects the type of infrastructure the module manages and is the main provider where it creates that infrastructure. The segment can contain additional hyphens. Examples: `terraform-google-vault` or `terraform-aws-ec2-instance`.
- The module repository must have a description which is used to populate the short description of the module. This should be a simple one sentence description of the module.
- The module must adhere to the standard module structure, `main.tf`, `variables.tf`, `outputs.tf`. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.
- x.y.z tags for releases. The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

### Reference

[Publishing to the Terraform Public Module Registry](#)





## Terraform Module Versions

Modules, like any piece of code, are never complete. There will always be new module requirements and changes.

Each distinct module address has associated with it a set of versions, each of which has an associated version number. Terraform assumes version numbers follow the Semantic Versioning 2.0 convention. Each module block may select a distinct version of a module, even if multiple blocks have the same source address.

- Task 1: Viewing Terraform Module Versions
- Task 2: Comparing Terraform Module Versions
- Task 3: Terraform Module Version Constraints

### Task 1: Viewing Terraform Module Versions

Each module in the Terraform Public registry is versioned. These versions syntactically must follow semantic versioning. The version argument can be specified as part of the module block as is shown in our example for the AWS VPC Terraform module.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.11.0"
}
```

When installing a module inside the working directory with a `terraform init` Terraform pulls down version of the module specified by the `version` argument.

```
terraform init

Initializing modules...
Downloading registry.terraform.io/terraform-aws-modules/vpc/aws 3.11.0 for
  vpc...
- vpc in .terraform/modules/vpc
```

### Task 2: Comparing Terraform Module Versions

Update the `vpc` module block to specify a particular module version, by adding the `version` argument. You can find the latest version of the module by viewing the module information in the Terraform Public Module Registry



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



### VPC Module

A screenshot of the HashiCorp Terraform Registry website. The top navigation bar includes the HashiCorp logo, 'Terraform Registry', a search bar ('Search Providers and Modules'), and links for 'Browse', 'Publish', and 'Sign-in'. The main content area shows the 'aws/vpc' module. It features the AWS logo, a brief description ('Terraform module which creates VPC resources on AWS'), and a list of versions: Version 3.11.0 (latest), Version 3.10.0, Version 3.9.0, Version 3.8.0, and Version 3.7.0. A 'Provision Instructions' box contains Terraform code to use the module. Below the module details are links for 'Readme', 'Inputs (166)', 'Outputs (109)', 'Dependency (1)', and 'Resources (76)'.

**Figure 1:** VPC Module Version

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "3.11.0"

  name = "my-vpc-terraform"
  cidr = "10.0.0.0/16"

  azs      = ["us-east-1a", "us-east-1b", "us-east-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Name      = "VPC from Module"
    Terraform = "true"
    Environment = "dev"
  }
}
```

Execute a `terraform init` followed by a `terraform plan` to validate that specifying the module version did not result in a change to the vpc.

Now we will change to an older version of the vpc module. Update the VPC module block to utilize version 1.73.0

```
module "vpc" {
```





```

source = "terraform-aws-modules/vpc/aws"
version = "1.73.0"

name = "my-vpc-terraform"
cidr = "10.0.0.0/16"

azs          = ["us-east-1a", "us-east-1b", "us-east-1c"]
private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
public_subnets = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

enable_nat_gateway = true
enable_vpn_gateway = true

tags = {
  Name      = "VPC from Module"
  Terraform = "true"
  Environment = "dev"
}
}
}

```

After changing the module version run a `terraform init` to install the 1.73.0 version of the module.

```

terraform init
Initializing modules...
Downloading registry.terraform.io/terraform-aws-modules/vpc/aws 1.73.0 for
  vpc...
- vpc in .terraform/modules/vpc

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock
  file
- Reusing previous version of hashicorp/tls from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/http from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock
  file
- Using previously-installed hashicorp/random v3.1.0
- Using previously-installed hashicorp/local v2.1.0
- Using previously-installed hashicorp/tls v3.1.0
- Using previously-installed hashicorp/aws v3.70.0
- Using previously-installed hashicorp/http v2.1.0

| Warning: Quoted references are deprecated
|   on .terraform/modules/vpc/main.tf line 110, in resource "

```





```
aws_route_table" "private":  
110: ignore_changes = ["propagating_vgws"]  
  
In this context, references are expected literally rather than in quotes  
. Terraform  
0.11 and earlier required quotes, but quoted references are now  
deprecated and will  
be removed in a future version of Terraform. Remove the quotes  
surrounding this  
reference to silence this warning.  
  
(and 2 more similar warnings elsewhere)
```

Notice that during the installation of this module version we received warnings that there are deprecated commands that this version of the module uses. Remember that Terraform modules are simply terraform configuration files and this version of the module is using terraform configuration that has been deprecated. This highlights why it is important to be sure we specify a version of a module that works within the Terraform core version we are using.

Run a `terraform validate` to showcase other errors surfaced with this older version of the module.

```
terraform validate  
  
Warning: Quoted references are deprecated  
  
  on .terraform/modules/vpc/main.tf line 110, in resource "aws_route_table" "private":  
110: ignore_changes = ["propagating_vgws"]  
  
In this context, references are expected literally rather than in quotes  
. Terraform 0.11 and  
earlier required quotes, but quoted references are now deprecated and  
will be removed in a future  
version of Terraform. Remove the quotes surrounding this reference to  
silence this warning.  
  
(and 2 more similar warnings elsewhere)  
  
Error: Error in function call  
  
  on .terraform/modules/vpc/main.tf line 26, in resource "aws_vpc" "this"  
  ::  
26:   tags = "${merge(map("Name", format("%s", var.name)), var.tags,  
var.vpc_tags)}"  
      |-----  
      | var.name will be known only after apply
```





```

| Call to function "map" failed: the "map" function was deprecated in
|   Terraform v0.12 and is no
| longer available; use tomap({ ... }) syntax to write a literal map.

| Error: Error in function call

|   on .terraform/modules/vpc/main.tf line 49, in resource "aws_vpc_dhcp_options" "this":
|     49:   tags = "${merge(map("Name", format("%s", var.name)), var.tags,
|       var.dhcp_options_tags)}"
|         |
|         |-----|
|         |   var.name will be known only after apply

| Call to function "map" failed: the "map" function was deprecated in
|   Terraform v0.12 and is no
| longer available; use tomap({ ... }) syntax to write a literal map.

| Error: Error in function call

|   on .terraform/modules/vpc/main.tf line 590, in resource "aws_eip" "nat"
|   ":
|     590:   tags = "${merge(map("Name", format("%s-%s", var.name, element(
|       var.azs, (var.single_nat_gateway ? 0 : count.index)))), var.tags, var.
|       nat_eip_tags)}"
|         |
|         |-----|
|         |   count.index is a number, known only after apply
|         |   var.azs will be known only after apply
|         |   var.name will be known only after apply
|         |   var.single_nat_gateway will be known only after apply

| Call to function "map" failed: the "map" function was deprecated in
|   Terraform v0.12 and is no
| longer available; use tomap({ ... }) syntax to write a literal map.

| Error: Error in function call

|   on .terraform/modules/vpc/main.tf line 711, in resource "aws_default_vpc" "this":
|     711:   tags = "${merge(map("Name", format("%s", var.default_vpc_name)), var.tags, var.default_vpc_tags)}"
|         |
|         |-----|
|         |   var.default_vpc_name will be known only after apply

| Call to function "map" failed: the "map" function was deprecated in
|   Terraform v0.12 and is no
| longer available; use tomap({ ... }) syntax to write a literal map.

```

It should now be obvious that this version of the VPC module is not compatible with our current





version of Terraform. Let's look at how we can incorporate module version constraints to make sure our code is using a compatible version of our module.

### Task 3: Terraform Module Version Constraints

When using modules installed from a module registry it is highly recommended to explicitly constrain the acceptable version numbers to avoid unexpected or unwanted changes. Terraform provides the ability to resolve any provided module version constraints and using them is highly recommended to avoid pulling in breaking changes. The `version` argument accepts a version constraint string. Terraform will use the newest installed version of the module that meets the constraint; if no acceptable versions are installed, it will download the newest version that meets the constraint.

Update the VPC module block to utilize any version greater than 3.0.0

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = ">3.0.0"

  name = "my-vpc-terraform"
  cidr = "10.0.0.0/16"

  azs           = ["us-east-1a", "us-east-1b", "us-east-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Name      = "VPC from Module"
    Terraform = "true"
    Environment = "dev"
  }
}
```

Run a `terraform init` to validate that the installed module meets the version constraints.

```
terraform init

Initializing modules...
Downloading registry.terraform.io/terraform-aws-modules/vpc/aws 3.11.0 for
  vpc...
- vpc in .terraform/modules/vpc
```

Version constraints are supported only for modules installed from a module registry, such as the public





Terraform Registry or Terraform Cloud's private module registry. Other module sources can provide their own versioning mechanisms within the source string itself, or might not support versions at all. In particular, modules sourced from local file paths do not support version; since they're loaded from the same source repository, they always share the same version as their caller.

Now that we have shown how to use modules from the Public Module Registry and constrain to appropriate versions we can cleanup from this section of the course by issuing a `terraform destroy` and removing the module references from the within our `main.tf`



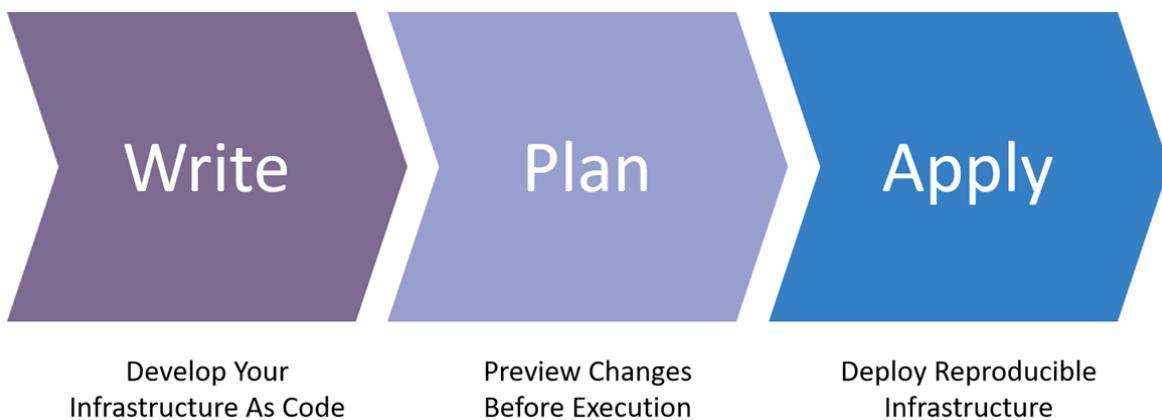


## Lab: Terraform Workflow

The core Terraform workflow has three steps:

1. Write - Author infrastructure as code.
2. Plan - Preview changes before applying.
3. Apply - Provision reproducible infrastructure.

## The Terraform Workflow



**Figure 1:** Terraform Workflow

The command line interface to Terraform is via the `terraform` command, which accepts a variety of subcommands such as `terraform init` or `terraform plan`. The Terraform command line tool is available for MacOS, FreeBSD, OpenBSD, Windows, Solaris and Linux.

- Task 1: Verify Terraform installation
- Task 2: Using the Terraform CLI
- Task 3: Initializing a Terraform Workspace
- Task 4: Generating a Terraform Plan
- Task 5: Applying a Terraform Plan
- Task 6: Terraform Destroy





## Task 1: Verify Terraform installation

Run the following command to check the Terraform version:

```
terraform -version
```

You should see:

```
Terraform v0.12.6
```

## Task 2: Using the Terraform CLI

You can get the version of Terraform running on your machine with the following command:

```
terraform -version
```

If you need to recall a specific subcommand, you can get a list of available commands and arguments with the help argument.

```
terraform -help
```

## Write

The Terraform language is Terraform's primary user interface. In every edition of Terraform, a configuration written in the Terraform language is always at the heart of the workflow.

## Task 3: Terraform Init

Initializing your workspace is used to initialize a working directory containing Terraform configuration files.

Copy the code snippet below into the file called `main.tf`. This snippet leverages the random provider, maintained by HashiCorp, to generate a random string.

```
main.tf
```

```
resource "random_string" "random" {
  length = 16
}
```





Once saved, you can return to your shell and run the `init` command shown below. This tells Terraform to scan your code and download anything it needs, locally.

```
terraform init
```

Once your Terraform workspace has been initialized you are ready to begin planning and provisioning your resources.

Note: You can validate that your workspace is initialized by looking for the presence of a `.terraform` directory. This is a hidden directory, which Terraform uses to manage cached provider plugins and modules, record which workspace is currently active, and record the last known backend configuration in case it needs to migrate state. This directory is automatically managed by Terraform, and is created during initialization.

#### Task 4: Generating a Terraform Plan

Terraform has a dry run mode where you can preview what Terraform will change without making any actual changes to your infrastructure. This dry run is performed by running a `terraform plan`.

In your terminal, you can run a plan as shown below to see the changes required for Terraform to reach your desired state you defined in your code. This is equivalent to running Terraform in a “dry” mode.

```
terraform plan
```

If you review the output, you will see 1 change will be made which is to generate a single random string.

Note: Terraform also has the concept of planning out changes to a file. This is useful to ensure you only apply what has been planned previously. Try running a plan again but this time passing an `-out` flag as shown below.

```
terraform plan -out myplan
```

This will create a plan file that Terraform can use during an `apply`.

#### Task 5: Applying a Terraform Plan

Run the command below to build the resources within your plan file.

```
terraform apply myplan
```





Once completed, you will see Terraform has successfully built your random string resource based on what was in your plan file.

Terraform can also run an `apply` without a plan file. To try it out, modify your `main.tf` file to create a random string with a length of 10 instead of 16 as shown below:

```
resource "random_string" "random" {  
    length = 10  
}
```

and run a `terraform apply`

```
terraform apply
```

Notice you will now see a similar output to when you ran a `terraform plan` but you will now be asked if you would like to proceed with those changes. To proceed enter `yes`.

Once complete the random string resource will be created with the attributes specified in the `main.tf` configuration file.

## Task 6: Terraform Destroy

The `terraform destroy` command is a convenient way to destroy all remote objects managed by a particular Terraform configuration. It does not delete your configuration file(s), `main.tf`, etc. It destroys the resources built from your Terraform code.

Run the command as shown below to run a planned destroy:

```
terraform plan -destroy
```

You will notice that it is planning to destroy your previously created resource. To actually destroy the random string you created, you can run a `destroy` command as shown below.

```
terraform destroy
```

Note: As similar to when you ran an `apply`, you will be prompted to proceed with the `destroy` by entering “yes”.

## Reference

The Core Terraform Workflow





## Lab: Initializing Terraform with `terraform init`

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control.

- Task 1: Initialize a Terraform working directory
- Task 2: Re-initialize after adding a new provider
- Task 3: Re-initialize after adding a new module
- Task 4: Re-initialize after modifying a Terraform backend
- Task 5: Other initialization steps/considerations

### Task 1: Initialize a Working Directory

You must run `terraform init` to initialize a new Terraform working directory. Copy the code snippet below into the file called `main.tf`. This snippet leverages the random provider, maintained by HashiCorp, to generate a random string.

`main.tf`

```
resource "random_pet" "server" {
  length = 2
}
```

Once saved, you can return to your shell and run the `init` command shown below. This tells Terraform to scan your code and download anything it needs, locally.

`terraform init`

Once your Terraform workspace has been initialized you are ready to begin planning and provisioning your resources.

Note: You can validate that your workspace is initialized by looking for the presence of a `.terraform` directory. This is a hidden directory, which Terraform uses to manage cached provider plugins and modules, record which workspace is currently active, and record the last known backend configuration in case it needs to migrate state. This directory is automatically managed by Terraform, and is created during initialization.





## Task 2: Re-initialize after add a new provider

You must run `terraform init` to initialize a new Terraform working directory, and after changing provider, module or backend configuration. You do not need to run `terraform init` before each command and it always safe to run multiple times.

Let's add anew provider for use within our `terraform.tf` file. Let's add the [Azure](#) provider:

`terraform.tf`

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "3.1.0"
    }
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.84.0"
    }
  }
}
```

Before re-initializing our working directory, let's verify that the [Azure](#) provider is not yet in use:

`terraform providers`

Providers required by configuration:

```
.
|-- provider[registry.terraform.io/hashicorp/aws]
|-- provider[registry.terraform.io/hashicorp/http] 2.1.0
```





## Hands-On Labs

```
|-- provider[registry.terraform.io/hashicorp/random] 3.1.0
|-- provider[registry.terraform.io/hashicorp/local] 2.1.0
|-- provider[registry.terraform.io/hashicorp/tls] 3.1.0
```

Now let's re-initialize our working directory to add the `azurerm` provider.

```
terraform init
```

Notice that the `azurerm` provider has now been installed and initialized.

```
Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock
  file
- Reusing previous version of hashicorp/local from the dependency lock
  file
- Reusing previous version of hashicorp/tls from the dependency lock file
- Finding hashicorp/azurerm versions matching "2.84.0"...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/http from the dependency lock file
- Installing hashicorp/azurerm v2.84.0...
- Installed hashicorp/azurerm v2.84.0 (signed by HashiCorp)
- Using previously-installed hashicorp/aws v3.63.0
- Using previously-installed hashicorp/http v2.1.0
- Using previously-installed hashicorp/random v3.1.0
- Using previously-installed hashicorp/local v2.1.0
- Using previously-installed hashicorp/tls v3.1.0
```

### Task 3: Re-initialize after adding a new module

During `init`, the configuration is searched for module blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments.

Add a Terraform module to your configuration.

```
main.tf
```

```
module "s3-bucket_example_complete" {
  source  = "terraform-aws-modules/s3-bucket/aws//examples/complete"
  version = "2.10.0"
}
```

In order for Terraform to use this module, it must first retrieve the module. This is done by running a `terraform init`.

Verify that the module has not yet been retrieved:





`terraform providers`

Retrieve the module by performing a `terraform init`

`terraform init`

```
Initializing modules...
Downloading terraform-aws-modules/s3-bucket/aws 2.10.0 for s3-
  bucket_example_complete...
- s3-bucket_example_complete in .terraform/modules/s3-
    bucket_example_complete/examples/complete
- s3-bucket_example_complete.cloudfront_log_bucket in .terraform/modules/
    s3-bucket_example_complete
- s3-bucket_example_complete.log_bucket in .terraform/modules/s3-
    bucket_example_complete
- s3-bucket_example_complete.s3_bucket in .terraform/modules/s3-
    bucket_example_complete
```

Validate that the module has been retrieved and initialized

`terraform providers`

Providers required by configuration:

```
-- provider[registry.terraform.io/hashicorp/tls] 3.1.0
-- provider[registry.terraform.io/hashicorp/azurerm] 2.84.0
-- provider[registry.terraform.io/hashicorp/aws]
-- provider[registry.terraform.io/hashicorp/http] 2.1.0
-- provider[registry.terraform.io/hashicorp/random] 3.1.0
-- provider[registry.terraform.io/hashicorp/local] 2.1.0
-- module.s3-bucket_example_complete
  |-- provider[registry.terraform.io/hashicorp/aws] >= 3.60.0
  |-- provider[registry.terraform.io/hashicorp/random] >= 2.0.0
  |-- module.cloudfront_log_bucket
    |-- provider[registry.terraform.io/hashicorp/aws] >= 3.50.0
  |-- module.log_bucket
    |-- provider[registry.terraform.io/hashicorp/aws] >= 3.50.0
  |-- module.s3_bucket
    |-- provider[registry.terraform.io/hashicorp/aws] >= 3.50.0
```

Re-running init with modules already installed will install the sources for any modules that were added to configuration since the last init, but will not change any already-installed modules. Use `-upgrade` to override this behavior, updating all modules to the latest available source code.

`terraform init -upgrade`

`Upgrading modules...`





```
Downloading terraform-aws-modules/s3-bucket/aws 2.10.0 for s3-
bucket_example_complete...
- s3-bucket_example_complete in .terraform/modules/s3-
  bucket_example_complete/examples/complete
- s3-bucket_example_complete.cloudfront_log_bucket in .terraform/modules/
  s3-bucket_example_complete
- s3-bucket_example_complete.log_bucket in .terraform/modules/s3-
  bucket_example_complete
- s3-bucket_example_complete.s3_bucket in .terraform/modules/s3-
  bucket_example_complete
```

## Task 4: Re-initialize after modifying changing a Terraform backend

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings. By default terraform uses a `local` backend and saves its state file to a `terraform.tfstate` file located in the working directory.

You can validate that your state file lives in the current directory by looking for the presence of a `terraform.tfstate` file. You may also see a backup that was created for this state file

```
-- main.tf
-- myplan
-- terraform.tf
-- terraform.tfstate
-- terraform.tfstate.backup
-- terraform.tfstate.d
|-- |-- development
|   |-- terraform.tfstate
|   |-- terraform.tfstate.backup
|-- variables.tf
```

You can modify the default backend that terraform uses by updating the terraform configuration block. Update the `terraform.tf` configuration to move terraform state to a different directory by including a `backend` block to the configuration.

```
terraform {
  backend "local" {
    path = "mystate/terraform.tfstate"
  }
}
```

Create a new subdirectory called `mystate` for terraform to use with the new backend.

```
terraform init
```





## Hands-On Labs

If your state file is empty then you would be presented with the following when re-initializing:

```
Initializing the backend...
```

```
Successfully configured the backend "local"! Terraform will automatically
use this backend unless the backend configuration changes.
```

If your state file is not empty then you would be presented with the following when re-initializing:

```
Initializing the backend...
```

```
Do you want to copy existing state to the new backend?
```

```
Pre-existing state was found while migrating the previous "local"
backend to the
newly configured "local" backend. No existing state was found in the
newly
configured "local" backend. Do you want to copy this state to the new "
local"
backend? Enter "yes" to copy and "no" to start with an empty state.
```

```
Enter a value: yes
```

```
Successfully configured the backend "local"! Terraform will automatically
use this backend unless the backend configuration changes.
```

Terraform will automatically copy the state information from the default location to your new directory during the re-initialization process.

```
.
|-- MyAWSKey.pem
|-- main.tf
|-- myplan
|-- mystate
|   |-- terraform.tfstate
|-- terraform.tf
|-- terraform.tfstate
|-- terraform.tfstate.backup
|-- variables.tf
```

When modifying terraform's backend you must re-initialize the working directory. Re-running init with an already-initialized backend will update the working directory to use the new backend settings. Either -reconfigure or -migrate-state must be supplied to update the backend configuration.

```
-reconfigure
```

```
Reconfigure the backend, ignoring any saved
configuration.
```

```
-migrate-state
any
```

```
Reconfigure the backend, and attempt to migrate
existing state.
```





Migrate the state to the `mystate` directory

```
terraform init -migrate-state
```

Now the state information (when the next `terraform apply` is run) will be updated in the `/mystate` directory.) Let's configure terraform back to the default current directory by removing the `backend` block within our `terraform.tf` and reinitializing.

```
terraform init -migrate-state
```

Initializing the backend...

Terraform has detected you're unconfiguring your previously set "local" backend.

Do you want to copy existing state to the new backend?

Pre-existing state was found while migrating the previous "local" backend to the

newly configured "local" backend. No existing state was found in the newly

configured "local" backend. Do you want to copy this state to the new "local"

backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value: yes

Successfully unset the backend "local". Terraform will now operate locally

.

Once successfully migrating back to the default you can delete the `mystate` directory.

## Task 5: Other initialization steps/considerations

For providers that are published in either the public Terraform Registry or in a third-party provider registry, `terraform init` will automatically find, download, and install the necessary provider plugins. If you cannot or do not wish to install providers from their origin registries, you can customize how Terraform installs providers using the provider installation settings in the CLI configuration.

To check the provider versions installed via `terraform init` run the `terraform version` command.

```
terraform version
```





```
Terraform v1.0.10
on darwin_amd64
+ provider registry.terraform.io/hashicorp/aws v3.64.2
+ provider registry.terraform.io/hashicorp/azurerm v2.84.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

You can modify the version line of any provider to be as specific or general as desired. Update the `version` line within your `terraform.tf` file for the `azurerm` provider installed earlier in the lab to an older provider version:

`terraform.tf`

```
azurerm = {
  source  = "hashicorp/azurerm"
  version = "2.57.0"
}
```

Utilize the `terraform init -upgrade` command to update the provider version for terraform to use. The `-upgrade` subcommand for `terraform init` will cause Terraform to take the newest available version matching the configured version constraints. If the `-upgrade` option is not issued Terraform will continue to use version installed or specified within the dependency lock file.

`terraform init`

```
Error: Failed to query available provider packages
```

```
Could not retrieve the list of available versions for provider hashicorp/azurerm: locked provider registry.terraform.io/hashicorp/azurerm 2.84.0 does not match configured version constraint 2.57.0; must use terraform init -upgrade to allow selection of new versions
```

`terraform init -upgrade`

- Installing hashicorp/azurerm v2.57.0...
- Installed hashicorp/azurerm v2.57.0 (signed by HashiCorp)

`terraform version`

```
Terraform v1.0.10
on darwin_amd64
+ provider registry.terraform.io/hashicorp/aws v3.64.2
+ provider registry.terraform.io/hashicorp/azurerm v2.57.0
```





```
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```





## Lab: Validate a Terraform configuration with `terraform validate`

The `terraform validate` command validates the configuration files in a directory, referring only to the Terraform configuration files. Validate runs checks that verify whether a configuration is syntactically valid and internally consistent.

- Task 1: Validate Terraform configuration
- Task 2: Terraform Validate False Positives
- Task 3: JSON Validation Output

### Task 1: Validate Terraform configuration

Validation requires an initialized working directory with any referenced plugins and modules installed. It is safe to run `terraform validate` at anytime.

Run a `terraform validate` against your Terraform configuration.

```
terraform validate
```

```
Success! The configuration is valid.
```

Make a change to the configuration that is not validate within Terraform and re-run a validate to see if the syntax error is picked up. We will include an argument in our resource block that isn't supported and see what `validate` reports.

Update the VPC resource block to include a `region` argument `main.tf`

```
resource "aws_vpc" "vpc" {
  region = "us-east-1"
  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
}
```

```
terraform validate
```

```
Error: Unsupported argument
```





```
on main.tf line 25, in resource "aws_vpc" "vpc":
25:   region = "us-east-1"
```

An argument named "region" is not expected here.

Notice that the `terraform validate` command will return the problem(s) along with file and line number in where they occur. Now replace the offending piece of code and re-issue a `terraform validate`.

```
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = "demo_environment"
    Terraform  = "true"
  }
}
```

`terraform validate`

Success! The configuration is valid.

## Task 2: Terraform Validate False Positives

The `terraform validate` commands runs checks that verify whether a configuration is syntactically valid and internally consistent, but it may not catch everything. We typically refer to these as false positives where the `terraform validate` reports successful but our plan/apply may still fail.

Let's look at a simple example with the configuration that stores our SSH Key using the Terraform `local` provider.

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "MyAWSKey.pem"
}
```

If we run a `terraform validate` against we will generate a success.





Success! The configuration is valid.

Now let's update the `local_file` resource to specify a file path that is incorrect/we don't have permissions to.

```
resource "tls_private_key" "generated" {
    algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
    content  = tls_private_key.generated.private_key_pem
    filename = "/bad_path/MyAWSKey.pem"
}
```

`terraform validate`  
Success! The configuration is valid.

`terraform apply`

```
tls_private_key.generated: Refreshing state... [id=502
de2c3e0a15eaf6e20a5a8cd616378daaafd62]
local_file.private_key_pem: Refreshing state... [id=81
a9356c0bca8224ea96ec7add0ac28eac442d07]
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
`-/+` destroy and then create replacement

Terraform will perform the following actions:

```
# local_file.private_key_pem must be replaced
-/+ resource "local_file" "private_key_pem" {
    ~ filename          = "MyAWSKey.pem" -> "/bad_path/MyAWSKey.pem"
        # forces replacement
    ~ id                = "81a9356c0bca8224ea96ec7add0ac28eac442d07"
        -> (known after apply)
        # (3 unchanged attributes hidden)
}
```

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only '`yes`' will be accepted to approve.

Enter a value: yes





```

local_file.private_key_pem: Destroying... [id=81
    a9356c0bca8224ea96ec7add0ac28eac442d07]
local_file.private_key_pem: Destruction complete after 0s
local_file.private_key_pem: Creating...

| Error: mkdir /bad_path: read-only file system

|   with local_file.private_key_pem,
|   on main.tf line 5, in resource "local_file" "private_key_pem":
|     5: resource "local_file" "private_key_pem" {
|
}

```

As you can see the execution of our configuration fails. This is because we don't have permission to create the key in the directory we specified. It is important to know that `terraform validate` will check that the HCL syntax is valid but not if the values provided for arguments are correct.

### Task 3: JSON Validation Output

You can also produce the output of a `terraform validate` in JSON format in the event you need to pass this data to another system.

```
terraform validate -json
```

```
{
  "format_version": "0.1",
  "valid": true,
  "error_count": 0,
  "warning_count": 0,
  "diagnostics": []
}
```

You can make the same invalid code change as performed in Task #1 to see how `terraform validate` returns an error in JSON format

```
terraform validate -json
```

```
{
  "format_version": "0.1",
  "valid": false,
  "error_count": 1,
  "warning_count": 0,
  "diagnostics": [
    {
      "severity": "error",

```





```
"summary": "Unsupported argument",
"detail": "An argument named \"region\" is not expected here.",
"range": {
    "filename": "main.tf",
    "start": {
        "line": 25,
        "column": 3,
        "byte": 511
    },
    "end": {
        "line": 25,
        "column": 9,
        "byte": 517
    }
},
"snippet": {
    "context": "resource \"aws_vpc\" \"vpc\"",
    "code": "  region = \"us-east-1\"",
    "start_line": 25,
    "highlight_start_offset": 2,
    "highlight_end_offset": 8,
    "values": []
}
}]}
```





## Lab: Generate and review an execution plan with `terraform plan`

The `terraform plan` command performs a dry-run of executing your terraform configuration and checks whether the proposed changes match what you expect before you apply the changes or share your changes with your team for broader review.

- Task 1: Generate and Review a plan
- Task 2: Save a Terraform plan
- Task 3: No Change Plans
- Task 4: Refresh Only plans

### Task 1: Generate and Review a plan

Prior to Terraform, users had to guess change ordering, parallelization, and rollout effect of their code. Terraform alleviates the guessing by creating a plan by performing the following steps:

- Reading the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing a set of change actions that should, if applied, make the remote objects match the configuration.

`terraform plan`

The plan shows you what will happen and the reasons for certain actions (such as re-create).





- + resource will be created
- resource will be destroyed
- ~ resource will be updated in-place
- /+ resources will be destroyed and re-created

**Figure 1:** Terraform Plan Resources

Terraform plans do not make any changes to your infrastructure they only report what will occur if a plan is executed against using a `terraform apply`.

## Task 2: Save a Terraform plan

The `terraform plan` command supports a number of different options, subcommands and customization options. These options can be reviewed using `terraform plan -help`.

```
terraform plan -help
```

You can save plans to a file to guarantee what will happen

```
terraform plan -out=myplan
```

```
.  
|-- MyAWSKey.pem  
|-- main.tf  
|-- myplan  
|-- terraform.tf  
|-- terraform.tfstate  
|-- terraform.tfstate.backup  
|-- variables.tf
```

You can also look at the details of a given plan file by running a `terraform show` command.





```
terraform show myplan
```

## Task 3: No Change Plans

As you can see the plan command within Terraofrm is extremly powerful for identifying any changes that need to occur against your configuration and the already deployed infrastructure. The plan command is also extremely powerful to validate that your real insfractructure and configuration are in sync. If this is true, the plan command will indicate that there are [No changes](#). This is sometimes referred to a a [No Change Plan](#) or [Zero Change Plan](#) and is extremly valuable.

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Because the [terraform plan](#) makes no changes to your infrastructure it can be run at any time without any penalty. In fact it should be run frequently to detect drift between your deployed infrastructure and configuration. A [No Change Plan](#) indicates that there is no drift.

The [terraform plan](#) should also be run after any terraform configuration has been refactored. Perhaps the configuration has been refactored to make use of variables or a different set of data constructs to make the code more reusable or easier to read. If these changes are merely to refactor code but should not result in any changes to your deployed infrastructure, a [No Change Plan](#) can be used to verify that the refactoring exercise were benign.

## Task 4: Refresh Only plans

The first step of the [terraform plan](#) process is to read the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date. It will then compare the current configuration to the prior state and note differences. If you wish to only perform the first part of this flow you can execute the plan with a [-refresh-only](#) option to make the

Change the tag of a given object inside the AWS Console





EC2 > Instances > i-0da540d207d546644 > Manage tags

### Manage tags Info

A tag is a custom label that you assign to an AWS resource. You can use tags to help organize and identify your instances.

Key	Value - optional	Remove
<input type="text" value="Name"/> X	<input type="text" value="Web EC2 Server"/> X	<button>Remove</button>
<input type="text" value="Provisioned"/> X	<input type="text" value="Terraform"/> X	<button>Remove</button>
<input type="text" value="Environment"/> X	<input type="text" value="default"/> X	<button>Remove</button>
<input type="text" value="Owner"/> X	<input type="text" value="Acme"/> X	<button>Remove</button>
<input type="text" value="Team"/> X	<input type="text" value="Engineering"/> X	<button>Remove</button>
<input type="text" value="Custom tag value"/>		
<a href="#">Add tag</a>		

You can add 45 more tags.

[Cancel](#) [Save](#)

**Figure 2:** AWS Tag

```
terraform plan -refresh-only
```

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the last "terraform apply":

```
# aws_instance.web_server has been changed
~ resource "aws_instance" "web_server" {
    id                               = "i-0da540d207d546644"
    ~ tags                           = {}
    + "Team" = "Engineering"
        # (1 unchanged element hidden)
}
~ tags_all                         = {}
+ "Team"      = "Engineering"
    # (4 unchanged elements hidden)
```





```
}
```

```
# (27 unchanged attributes hidden)
```

```
# (5 unchanged blocks hidden)
```

```
}
```

This is a refresh-only plan, so Terraform will not take any actions to undo these. If you were expecting these changes then you can apply **this** plan to record the updated values in the Terraform state without changing any remote objects.

Terraform's Refresh-only mode goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform. This can be useful if you've intentionally changed one or more remote objects outside of the usual workflow (e.g. while responding to an incident) and you now need to reconcile Terraform's records with those changes.





## Lab: Execute changes to infrastructure with Terraform `terraform apply`

The `terraform apply` command executes the actions proposed in a Terraform plan.

- Task 1: Apply a Terraform Plan
- Task 2: Auto Approve Execution of an Apply
- Task 3: Execute a saved Terraform Plan

### Task 1: Apply a Terraform Plan

The most straightforward way to use `terraform apply` is to run it without any arguments at all, in which case it will automatically create a new execution plan (as if you had run `terraform plan`) and then prompt you to approve that plan, before taking the indicated actions.

Make an update the `Environment` tag of the `vpc` resource

```
resource "aws_vpc" "vpc" {
    cidr_block = var.vpc_cidr
    tags = {
        Name      = var.vpc_name
        Environment = "stage"
        Terraform   = "true"
    }
}
```

```
terraform apply
```

Review the new execution plan and approve it.

```
Terraform will perform the following actions:
# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
    id                               = "vpc-08b492b03641cb916"
    ~ tags                           = {}
    ~ "Environment" = "demo_environment" -> "stage"
    # (2 unchanged elements hidden)
}
~ tags_all                         = {}
~ "Environment" = "demo_environment" -> "stage"
# (4 unchanged elements hidden)
```





```

}
# (14 unchanged attributes hidden)
}

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

aws_vpc.vpc: Modifying... [id=vpc-08b492b03641cb916]
aws_vpc.vpc: Modifications complete after 1s [id=vpc-08b492b03641cb916]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

```

## Task 2: Auto Approve Execution of an Apply

Make another update the `Environment` tag of the `vpc` resource

```

resource "aws_vpc" "vpc" {

  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = "QA"
    Terraform   = "true"
  }
}

```

```
terraform apply -auto-approve
```

Terraform will perform the following actions:

```

# aws_vpc.vpc will be updated in-place
~ resource "aws_vpc" "vpc" {
  id                               = "vpc-08b492b03641cb916"
  ~ tags                           = {
    ~ "Environment" = "stage" -> "QA"
    # (2 unchanged elements hidden)
  }
  ~ tags_all                      = {
    ~ "Environment" = "stage" -> "QA"
  }
}

```





```

        # (4 unchanged elements hidden)
    }
    # (14 unchanged attributes hidden)
}

Plan: 0 to add, 1 to change, 0 to destroy.
aws_vpc.vpc: Modifying... [id=vpc-08b492b03641cb916]
aws_vpc.vpc: Modifications complete after 1s [id=vpc-08b492b03641cb916]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

```

Note that there is no prompt for approval of the plan with this execution. This is ideal for automated pipelines and workflows but should be used with caution.

### Task 3: Execute a saved Terraform Plan

Make another update the `Environment` tag of the `vpc` resource

```
resource "aws_vpc" "vpc" {

  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = "test-dev"
    Terraform   = "true"
  }
}
```

```
terraform plan -out=myplan
```

```
terraform apply myplan
```

```

aws_vpc.vpc: Modifying... [id=vpc-08b492b03641cb916]
aws_vpc.vpc: Modifications complete after 2s [id=vpc-08b492b03641cb916]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

```





## Lab: Terraform Destroy

You've now learned about all of the key features of Terraform, and how to use it to manage your infrastructure. The last command we'll use is `terraform destroy`. The `terraform destroy` command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.

- Task 1: Destroy your infrastructure

### Task 1: Destroy your infrastructure

The `terraform destroy` command destroys all remote objects managed by a particular Terraform configuration. It does not delete your configuration file(s), `main.tf`, etc. It destroys the resources built from your Terraform code.

Run the command `terraform destroy`:

```
terraform destroy
```

```
Plan: 0 to add, 0 to change, 39 to destroy.
```

```
Do you really want to destroy all resources?
```

```
Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only 'yes' will be accepted to confirm.
```

```
Enter a value: yes
```

```
...
```

```
Destroy complete! Resources: 39 destroyed.
```

You'll need to confirm the action by responding with `yes`. You could achieve the same effect by removing all of your configuration and running `terraform apply`, but you often will want to keep the configuration, but not the infrastructure created by the configuration.

You could also execute a `terraform apply` with a `-destroy` flag to achieve the same results.

```
terraform apply -destroy
```

While you will typically not want to destroy long-lived objects in a production environment, Terraform is sometimes used to manage ephemeral infrastructure for development purposes, in which case you can use `terraform destroy` to conveniently clean up all of those temporary objects once you are finished with your work.





## Lab: Terraform State Default Local Backend

In order to correctly manage your infrastructure resources, Terraform stores and operates on the state of your managed infrastructure. Terraform uses this state on each execution to plan and make changes. This state must be stored and maintained on each execution so future operations can be performed correctly. The location and method of operation of Terraform's state is determined by the Terraform [backend](#). By default Terraform uses a `local` backend, where state information is stored and acted upon locally within the working directory in a local file named `terraform.tfstate`.

- Task 1: Show current state (CLI Operation)
- Task 2: Show state file location
- Task 3: View/Update Terraform local backend configuration
- Task 4: Modify, Plan and Execute Changes
- Task 5: Show New State and State Backup

### Task 1: Show Current State via CLI

On your workstation, navigate to the /workstation/terraform directory. To view the applied configuration utilize the `terraform show` command to view the resources created and find the IP address for your instance.

```
terraform show
```

Note: If this command doesn't yield any information then you will need to redeploy your VPC infrastructure following the steps in Objective 1b via a `terraform apply`.

Alternatively you can run a `terraform state list` to list all of the items in Terraform's managed state.

```
terraform state list
```

```
data.aws_ami.ubuntu
data.aws_availability_zones.available
data.aws_region.current
aws_eip.nat_gateway_eip
aws_instance.ubuntu_server
aws_instance.web_server
aws_internet_gateway.internet_gateway
aws_key_pair.generated
aws_nat_gateway.nat_gateway
```





```

aws_route_table.private_route_table
aws_route_table.public_route_table
aws_route_table_association.private["private_subnet_1"]
aws_route_table_association.private["private_subnet_2"]
aws_route_table_association.private["private_subnet_3"]
aws_route_table_association.public["public_subnet_1"]
aws_route_table_association.public["public_subnet_2"]
aws_route_table_association.public["public_subnet_3"]
aws_security_group.ingress-ssh
aws_security_group.vpc-ping
aws_security_group.vpc-web
aws_subnet.private_subnets["private_subnet_1"]
aws_subnet.private_subnets["private_subnet_2"]
aws_subnet.private_subnets["private_subnet_3"]
aws_subnet.public_subnets["public_subnet_1"]
aws_subnet.public_subnets["public_subnet_2"]
aws_subnet.public_subnets["public_subnet_3"]
aws_vpc.vpc
local_file.private_key_pem
random_string.random
tls_private_key.generated

```

## Task 2: Show state file location

The next logical question is “where is state stored”? By default, Terraform stores state locally in a JSON file on disk, but it also supports a number of state backends for storing “remote state”. If you are still learning how to use Terraform, we recommend using the default `local` backend, which requires no configuration.

Terraform’s `local` state is stored on disk as JSON, and that file must always be up to date before a person or process runs Terraform. If the state is out of sync, the wrong operation might occur, causing unexpected results.

By default terraform uses a `local` backend and saves its state file to a `terraform.tfstate` file located in the working directory. You can validate that your state file lives in the current directory by looking for the presence of a `terraform.tfstate` file. You may also see a backup that was created for this state file

```

|-- main.tf
|-- MyAWSKey.pem
|-- terraform.tf
|-- terraform.tfstate
|-- terraform.tfstate.backup

```





```

|-- terraform.tfstate.d
|-- |-- development
    |-- terraform.tfstate
    |-- terraform.tfstate.backup
|-- variables.tf

```

If you open and view the `terraform.tfstate` file you will notice that it is stored in a `json` format. As discussed in previous labs it is never a good idea to modify the contents of this file directly, but rather use the standard Terraform workflow and CLI state options available for advanced Terraform state management.

### Task 3: View/Update Terraform local backend configuration

The terrafrom backend end configuration for a given working directory is specified in the Terraform configuration block. Our terraform configuration block for this lab is located in the `terraform.tf` file.

```

terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "3.1.0"
    }
  }
}

```

By default there is no `backend` configuration block within our configuration. Because no `backend` was included in our configuration Terraform will use it's default backend - `local`. This is why we see





the `terraform.tfstate` file in our working directory. If we want to be explicit about which backend Terraform should use it would cause no harm to add the following to our Terraform configuration block within the `terraform.tfstate` file

```
terraform {
  backend "local" {
    path = "terraform.tfstate"
  }
}
```

This is not required and generally not performed as it is the Terraform defalt backend that `terraform` uses.

Note: Be sure not to copy just the `backend` block above and not the full `terraform` block. You can validate the syntax is correct by issuing a `terraform validate`

Anytime a new piece of configuration is added to a Terraform configuration block the working directory must be re-initialized.

```
terraform init
```

#### Task 4: Modify, Plan and Execute Changes

During execution, Terraform will examine the state of the currently running infrastructure, determine what differences exist between the current state and the revised desired state, and indicate the necessary changes that must be applied. When approved to proceed, only the necessary changes will be applied, leaving existing, valid infrastructure untouched. Terraform can perform in-place updates after changes are made to the `main.tf` configuration file. Update your `main.tf` to include an second EC2 instance in the public subnet 2:

Append the following code to `main.tf`

```
# Terraform Resource Block - To Build EC2 instance in Public Subnet
resource "aws_instance" "web_server_2" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnets["public_subnet_2"].id
  tags = [
    Name = "Web EC2 Server"
  ]
}
```





## Hands-On Labs

Save the configuration. Plan and apply the changes you just made and note the output differences for additions, deletions, and in-place changes.

Run a `terraform plan` to see the updates that Terraform needs to make.

```
terraform plan
```

Terraform used the selected providers to generate the following execution plan.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
# aws_instance.web_server will be created
+ resource "aws_instance" "web_server_2" {
    + ami                                     = "ami-083654bd07b5da81d"
    + arn                                     = (known after apply)
    + associate_public_ip_address             = (known after apply)
    + availability_zone                      = (known after apply)
    + cpu_core_count                         = (known after apply)
    + cpu_threads_per_core                  = (known after apply)
    + disable_api_termination               = (known after apply)
    + ebs_optimized                          = (known after apply)
    + get_password_data                     = false
    + host_id                                = (known after apply)
    + id                                      = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_state                          = (known after apply)
    + instance_type                           = "t2.micro"
    + ipv6_address_count                    = (known after apply)
    + ipv6_addresses                         = (known after apply)
    + key_name                               = (known after apply)
    + monitoring                             = (known after apply)
    + outpost_arn                            = (known after apply)
    + password_data                          = (known after apply)
    + placement_group                       = (known after apply)
    + placement_partition_number            = (known after apply)
    + primary_network_interface_id          = (known after apply)
    + private_dns                            = (known after apply)
    + private_ip                             = (known after apply)
    + public_dns                            = (known after apply)
    + public_ip                             = (known after apply)
    + secondary_private_ips                = (known after apply)
    + security_groups                       = (known after apply)
    + source_dest_check                     = true
    + subnet_id                             = "subnet-0ed3366fd5647c0e7"
    + tags                                    = {
```





```

        + "Name" = "Web EC2 Server"
    }
+ tags_all = {
    + "Name" = "Web EC2 Server"
}
+ tenancy = (known after apply)
+ user_data = (known after apply)
+ user_data_base64 = (known after apply)
+ vpc_security_group_ids = (known after apply)

+ capacity_reservation_specification {
    + capacity_reservation_preference = (known after apply)

    + capacity_reservation_target {
        + capacity_reservation_id = (known after apply)
    }
}

+ ebs_block_device {
    + delete_on_termination = (known after apply)
    + device_name = (known after apply)
    + encrypted = (known after apply)
    + iops = (known after apply)
    + kms_key_id = (known after apply)
    + snapshot_id = (known after apply)
    + tags = (known after apply)
    + throughput = (known after apply)
    + volume_id = (known after apply)
    + volume_size = (known after apply)
    + volume_type = (known after apply)
}

+ enclave_options {
    + enabled = (known after apply)
}

+ ephemeral_block_device {
    + device_name = (known after apply)
    + no_device = (known after apply)
    + virtual_name = (known after apply)
}

+ metadata_options {
    + http_endpoint = (known after apply)
    + http_put_response_hop_limit = (known after apply)
    + http_tokens = (known after apply)
}

+ network_interface {
}

```





```

+ delete_on_termination = (known after apply)
+ device_index          = (known after apply)
+ network_interface_id = (known after apply)
}

+ root_block_device {
    + delete_on_termination = (known after apply)
    + device_name           = (known after apply)
    + encrypted              = (known after apply)
    + iops                   = (known after apply)
    + kms_key_id             = (known after apply)
    + tags                   = (known after apply)
    + throughput              = (known after apply)
    + volume_id               = (known after apply)
    + volume_size              = (known after apply)
    + volume_type              = (known after apply)
}
}

Plan: 1 to add, 0 to change, 0 to destroy.

```

Due to the stateful nature of Terraform, it only needs to add the one additional resource to our infrastructure build out.

### Step 3.1.2

Run a `terraform apply` to execute the updates that Terraform needs to make.

```
terraform apply
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value:
```

When prompted to apply the changes, respond with `yes`.

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```





```
aws_instance.web_server_2: Creating...
aws_instance.web_server_2: Still creating... [10s elapsed]
aws_instance.web_server_2: Still creating... [20s elapsed]
aws_instance.web_server_2: Still creating... [30s elapsed]
aws_instance.web_server_2: Creation complete after 33s [id=i-0331d9d6ff38214a3]
```

Apply **complete!** Resources: 1 added, 0 changed, 0 destroyed.

## Task 5: Show New State and State Backup

To view the applied configuration utilize the `terraform state list` command to view the resources created. Look for the `aws_instance.web_server_2` which is now present within Terraform's managed state. You can then show the details of this resource by running a `terraform state show aws_instance.web_server_2`

```
terraform state list
terraform state show aws_instance.web_server_2
```

Terraform State example:

```
# aws_instance.web_server_2:
resource "aws_instance" "web_server" {
  ami                               = "ami-036490d46656c4818"
  arn                               = "arn:aws:ec2:us-east-1:508140242758:instance/i-0d544e90777ca8c2f"
  associate_public_ip_address       = true
  availability_zone                 = "us-east-1b"
  cpu_core_count                   = 1
  cpu_threads_per_core            = 1
  disable_api_termination         = false
  ebs_optimized                    = false
  get_password_data                = false
  hibernation                      = false
  id                               = "i-0d544e90777ca8c2f"
  instance_initiated_shutdown_behavior = "stop"
  instance_state                  = "running"
  instance_type                   = "t2.micro"
  ipv6_address_count              = 0
  ipv6_addresses                  = []
  monitoring                      = false
  primary_network_interface_id    = "eni-0445ae3a8b38ae47a"
  private_dns                      = "ip-10-0-101-117.ec2.internal"
  private_ip                       = "10.0.101.117"
  public_ip                        = "18.234.248.120"
  secondary_private_ips           = []
```





```

security_groups                  = []
source_dest_check               = true
subnet_id                       = "subnet-0e3cbf2e577579360"
tags                            = {
    "Name" = "Ubuntu EC2 Server"
}
tags_all                        = {
    "Name" = "Ubuntu EC2 Server"
}
tenancy                          = "default"
vpc_security_group_ids          = [
    "sg-097b59a05720fb97c",
]

capacity_reservation_specification {
    capacity_reservation_preference = "open"
}

credit_specification {
    cpu_credits = "standard"
}

enclave_options {
    enabled = false
}

metadata_options {
    http_endpoint           = "enabled"
    http_put_response_hop_limit = 1
    http_tokens             = "optional"
}

root_block_device {
    delete_on_termination = true
    device_name          = "/dev/sda1"
    encrypted             = false
    iops                  = 100
    tags                  = {}
    throughput             = 0
    volume_id              = "vol-053758fb913734c4c"
    volume_size            = 8
    volume_type            = "gp2"
}
}

```

Since the state of our configuration has changed, Terraform by default keeps a backup copy of the state in a `terraform.tfstate.backup` file.





```
|-- terraform.tfstate  
|-- terraform.tfstate.backup
```





## Lab: Terraform State Locking

Terraform uses persistent state data to keep track of the resources it manages. Since it needs the state in order to know which real-world infrastructure objects correspond to the resources in a configuration, everyone working with a given collection of infrastructure resources must be able to access the same state data.

Terraform's local state is stored on disk as JSON, and that file must always be up to date before a person or process runs Terraform. If the state is out of sync, the wrong operation might occur, causing unexpected results. If supported, the state backend will "lock" to prevent concurrent modifications which could cause corruption.

This lab demonstrates Terraform State locking.

- Task 1: Update Terraform Configuration
- Task 2: Issue a Terraform Apply
- Task 3: Specify a Terraform Lock Timeout
- Task 4: Explore State Backends that Support Locking

### Task 1: Update Terraform Configuration

Update the `main.tf` file and change the `tags` value of your `aws_instance.web_server_2` block

```
resource "aws_instance" "web_server_2" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnets["public_subnet_2"].id
  tags = {
    Name = "Web EC2 Server 2"
  }
}
```

### Task 2: Generate a Terraform State Lock

This task requires that you open two separate terminal windows to your working directory.

In Terminal #1 generate a lock on your state file by issuing a `terraform apply`

`Terminal #1`





```
terraform apply
```

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

Do not provide any answer at this time.

In Terminal #2 within the same working directory, issue a `terraform apply`

Terminal #2

```
terraform apply
```

Error: Error acquiring the state lock  
  
Error message: resource temporarily unavailable  
Lock Info:  
ID: 7aa0f3c3-51dc-f0d8-76cf-64f4953cbfff  
Path: terraform.tfstate  
Operation: OperationTypeApply  
Who: gabe@MacBook-Pro.local  
Version: 1.0.10  
Created: 2021-11-08 04:56:27.40246 +0000 UTC  
Info:

Terraform acquires a state lock to protect the state from being written by multiple users at the same time. Please resolve the issue above and **try** again. For most commands, you can disable locking with the "`-lock=false`" flag, but **this** is not recommended.

### Task 3: Specify a Terraform Lock Timeout

In Terminal #2, re-run a `terraform apply` this time with `-lock-timeout` value

Terminal #2

```
terraform apply -lock-timeout=60s  
Acquiring state lock. This may take a few moments...
```





Terminal #1 Answer `no` to the `terraform apply` command in the first terminal to free the state lock and observe behavior in the second terminal.

## Task 4: Explore State Backends that Support Locking

Not all Terraform backends support locking - Terraform's documentation identifies which backends support this functionality. Some common Terraform backends that support locking include:

- Remote Backend (Terraform Enterprise, Terraform Cloud)
- AWS S3 Backend (with DynamoDB)
- Google Cloud Storage Backend
- Azure Storage Backend

Obviously locking is an important feature of a Terraform backend in which there are multiple people collaborating on a single state file.





## Lab: Terraform State Backend Authentication

The `local` backend stores state as a local file on disk, but other backend types store state in a remote service of some kind, which allows multiple people to access it. Accessing state in a remote service generally requires some kind of access credentials since state data contains extremely sensitive information. It is important to strictly control who can access your Terraform backend.

We will look at two different backend types compatible with Terraform and how each handles authentication.

- Task 1: Authentication: S3 Standard Backend
- Task 2: Authentication: Remote Enhanced Backend

## Backend Configuration: Authentication

Some backends allow us to provide access credentials directly as part of the configuration. However, in normal use we do not recommend including access credentials as part of the backend configuration. Instead, leave those arguments completely unset and provide credentials via the credentials files or environment variables that are conventional for the target system, as described in the documentation for each backend.

## Task 1: Authentication: S3 Standard Backend

The `terraform` backend end configuration for a given working directory is specified in the Terraform configuration block. Our `terraform` configuration block for this lab is located in the `terraform.tf` file.

The `s3` backend stores Terraform state as a given key in a given bucket on Amazon S3. This backend supports several different methods in which the Terraform CLI can authenticate to the Amazon S3 bucket.

### Step 1.1 - Create the bucket in AWS

Create a new bucket within AWS to centrally store your `terraform` state files:





Amazon S3 > Create bucket

## Create bucket Info

Buckets are containers for data stored in S3. [Learn more](#)

### General configuration

Bucket name

Bucket name must be unique and must not contain spaces or uppercase letters. [See rules for bucket naming](#)

AWS Region

US East (N. Virginia) us-east-1

Copy settings from existing bucket - *optional*  
Only the bucket settings in the following configuration are copied.

[Choose bucket](#)

### Block Public Access settings for this bucket

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#)

- Block all public access**  
Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.
- Block public access to buckets and objects granted through new access control lists (ACLs)**

**Figure 1:** S3 Remote State Storage

If you're using S3 as a backend, you'll want to configure an IAM policy that solely grants access to the S3 bucket for production to a small handful of trusted people or perhaps solely just the CI server you use to deploy to your environments.

### Step 1.2 - Remove existing resources with `terraform destroy`

If you already have a state file present with infrastructure deployed from previous labs we will first issue a cleanup of our infrastructure using a `terraform destroy` before changing our our backend.





This is not a requirement as Terraform supports the migration of state data between backends, which will be covered in a future lab.

```
terraform destroy

Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes
```

### Step 1.2 - Update Terraform Configuration to use s3 backend

Update the `terraform.tf` configuration to utilize the `s3` backend with the required arguments.

```
terraform {
  backend "s3" {
    bucket = "myterraformstate"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
}
```

Example:

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state-ghm"
    key    = "prod/aws_infra"
    region = "us-east-1"
  }
}
```

Note: A Terraform configuration can only specify a single backend. If a backend is already configured be sure to replace it. Copy just the `backend` block above and not the full `terraform` block. You can validate the syntax is correct by issuing a `terraform validate`

### Step 1.3 - Provide Terraform AWS credentials to connect to S3 Bucket

Providing credentials for accessing state in an S3 bucket can be done in a number of different ways. This lab will showcase using environment variables but a shared credentials file can also be used. It is important to protect any credential information so while it is possible to set these values in the code itself it is strongly not recommended.





Source credentials to the S3 backend using the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables.

```
export AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
export AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
```

(Optional): If using Multi-Factor Authentication you can source the token using the `AWS_SESSION_TOKEN` environment variable.

(Optional): In lieu of setting environment variables you can also utilize an AWS shared credentials file by specifying the path to the file using the `shared_credentials_file` argument within the backend configuration block. This defaults to `~/.aws/credentials`.

#### Step 1.4 - Verify Authentication to S3 Backend

Once the configuration is complete, you can verify authentication to the S3 backend by first removing infrastructure that has already been deployed with a `terraform destroy` and performing a `terraform init`

```
terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

#### Step 1.5 - Write Terraform State to S3 Backend

Now that authentication has been verified, we can build out the infrastructure with our S3 backend for storing state we can issue a `terraform apply`

```
terraform apply

...
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
```





Amazon S3 > my-terraform-state-ghm > prod/

prod/

**Objects** Properties

**Objects (1)**

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others

[\[View\]](#)

[C](#) [Copy S3 URI](#) [Copy URL](#) [Download](#) [Open](#) [Delete](#) [Actions ▾](#)

[Find objects by prefix](#)

<input type="checkbox"/>	Name	Type	Last modified
<input type="checkbox"/>	<a href="#">aws_infra</a>	-	

**Figure 2:** S3 Backend Object

Now the state file for your infrastructure build out is stored remotely on the S3 object store in your bucket. This can now be utilized by others who have appropriate permissions to the S3 bucket as we have successfully centralized the terraform state file.

#### Step 1.6 - Remove Terraform AWS credentials to connect to S3 Bucket

If you would like to remove access to a centralized state file, you can modify the credentials to your S3 bucket. To showcase unauthenticated access, let's change the source credentials to the S3 backend by incorrectly setting the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables.

```
export AWS_ACCESS_KEY_ID="notvalid"
export AWS_SECRET_ACCESS_KEY="notvalid"
```

```
terraform state list
```





```
Error: error configuring S3 Backend: error validating provider credentials
  : error calling sts:GetCallerIdentity: InvalidClientTokenId: The
    security token included in the request is invalid
    status code: 403, request id: 00771614-5553-4eaf-ae8d-a3ce54c66060
```

Once we change these back, you can see we again have access.

```
export AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
export AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
```

```
terraform state list
```

## Task 2: Authentication: Remote Enhanced Backend

The Terraform remote backend stores Terraform state and may be used to run operations in Terraform Cloud. This backend supports the ability to store Terraform state information and perform operations all within Terraform Cloud, based on privileged access.

### Step 2.1

For this task you will have to sign up for a Terraform Cloud account. In order to store state remotely on Terraform Cloud using the `remote` backend we need to create a user token and configure our local environment to utilize that token. We will leverage the `terraform login` command to obtain and save an API token for authenticating to Terraform Cloud.

### Step 2.2

**Note:** You can skip this step if you've already created a organization.

Log in to Terraform Cloud and go to the new organization page:

- New users are automatically taken to the new organization page.
- If your user account is already a member of an organization, open the organization switcher menu in the top navigation bar and click the “Create new organization” button.

Enter a unique organization name and an email address for notifications, then click the “Create organization” button.





## Create a new organization

Organizations are privately shared spaces for teams to collaborate on infrastructure. [Learn more](#) about organizations in Terraform Cloud.

**Organization name**

e.g. company-name

Organization names must be unique and will be part of your resource names used in various tools, for example `hashicorp/www-prod`.

**Email address**

gabe@maentz.net

The organization email is used for any future notifications, such as billing alerts, and the organization avatar, via [gravatar.com](#).

**Create organization**

**Figure 3:** New Organization

### Step 2.3

Terraform's CLI needs credentials before it can access Terraform Cloud. We will leverage the `terraform login` command to perform our login to TFC.

`terraform login`

Terraform will request an API token `for app.terraform.io` using your browser.

If `login` is successful, Terraform will store the token `in` plain text `in` the following file `for` use by subsequent commands:  
`/Users/gabe/.terraform.d/credentials.tfrc.json`

Do you want to proceed?  
Only '`yes`' will be accepted to confirm.

Enter a value: yes





Open the following URL to access the tokens page **for** app.terraform.io:  
<https://app.terraform.io/app/settings/tokens?source=terraform-login>

---

Generate a token using your browser, and copy-paste it into this prompt.

Terraform will store the token **in** plain text **in** the following file  
**for** use by subsequent commands:  
`/home/nyl/.terraform.d/credentials.tfrc.json`

Token **for** app.terraform.io:  
Enter a value:

Retrieved token **for** user gabe\_maentz

---

Success! Terraform has obtained and saved an API token.

The new API token will be used **for** any future Terraform **command** that must  
make  
authenticated requests to app.terraform.io.





**Create API token**

Choose a description to help you identify this token later.

**Description**

**Create API token** **Cancel**

**Figure 4:** TFC\_TOKEN

Generate a token using your browser, and copy-paste it into this prompt.

**Create API token**

Your new API token is displayed below.

Pj249SXwubLWsw.atlasv1.vhSygiVnve10PvbtyRSUoAXsJ0JC0x6rTcX1F3C  
uj0yo4SjPUhpaRzrkFKnUdnb60Y

Copied!

Click on the token to copy it, and paste into your Terraform login prompt to continue.

**Done**

**Figure 5:** TFC\_TOKEN

Token **for** app.terraform.io:  
Enter a value:





```
Retrieved token for user gabe_maentz
```

---

```
Welcome to Terraform Cloud!
```

```
Documentation: terraform.io/docs/cloud
```

```
New to TFC? Follow these steps to instantly apply an example  
configuration:
```

```
$ git clone https://github.com/hashicorp/tfc-getting-started.git  
$ cd tfc-getting-started  
$ scripts/setup.sh
```

We will leverage this `remote` backend authentication to interact with Terraform Cloud from the CLI of our working directory in future labs.





## Lab: Terraform State Backend Storage

In order to properly and correctly manage your infrastructure resources, Terraform stores the state of your managed infrastructure. Each Terraform configuration can specify a backend which defines exactly where and how operations are performed. Most backends support security and collaboration features so using a backend is a must-have both from a security and teamwork perspective.

Terraform has a built-in selection of backends, and the configured backend must be available in the version of Terraform you are using. In this lab we will explore the use of some common Terraform standard and enhanced backends.

- Task 1: Standard Backend: S3
- Task 2: Standard Backend: HTTP Backend (Optional)

### Standard Backends

The built in Terraform standard backends store state remotely and perform terraform operations locally via the command line interface. Popular standard backends include:

- AWS S3 Backend (with DynamoDB)
- Google Cloud Storage Backend
- Azure Storage Backend

Consult Terraform documentation for a full list of Terraform standard backends

Most backends also support collaboration features so using a backend is a must-have both from a security and teamwork perspective. Not all of these features need to be configured and enabled, but we will walk you through some of the most beneficial items including versioning, encryption and state locking.

### Task 1: Standard Backend: S3

#### Step 1.1 - Create S3 Bucket and validate Terraform Configuration

In the previous lab we created an S3 bucket to centralize our terraform state to a remote S3 bucket. If you have not performed these steps please step through them as outlined in the previous lab. If you have already performed these steps you can move to the next step of the lab.





### Step 1.2 - Validate State on S3 Backend

We will want to validate that we can authenticate to our terraform backend and list the information in the state file. Let's validate that our configuration is pointing to the correct bucket and path.

`terraform.tf`

```
terraform {  
  backend "s3" {  
    bucket = "myterraformstate"  
    key    = "path/to/my/key"  
    region = "us-east-1"  
  }  
}
```

```
terraform state list
```

### Step 1.3 - Enable Versioning on S3 Bucket

Enabling versioning on our terraform backend is important as it allows us to restore the previous version of state should we need to. The `s3` backend supports versioning, so every revision of your state file is stored.





Amazon S3 > my-terraform-state-ghm > Edit Bucket Versioning

## Edit Bucket Versioning Info

**Bucket Versioning**

Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. [Learn more](#)

**Bucket Versioning**

- Suspend**  
This suspends the creation of object versions for all operations but preserves any existing object versions.
- Enable**

**Info** After enabling Bucket Versioning, you might need to update your lifecycle rules to manage previous versions of objects.

**Multi-factor authentication (MFA) delete**  
An additional layer of security that requires multi-factor authentication for changing Bucket Versioning settings and permanently deleting object versions. To modify MFA delete settings, use the AWS CLI, AWS SDK, or the Amazon S3 REST API. [Learn more](#)

Disabled

**Cancel** **Save changes**

**Figure 1:** S3 Versioning

Once versioning is enabled on your bucket let's make a configuration change that will result in a state change and execute that change with a `terraform apply`.

Update the size of your web server from `t2.micro` to a `t2.small` and apply the change.

```
resource "aws_instance" "web_server_2" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.small"
  subnet_id     = aws_subnet.public_subnets["public_subnet_2"].id
  tags = [
    { Name = "Web EC2 Server 2" }
  ]
}
```

```
terraform apply
```



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



Now you can see that your state file has been updated and if you check [Show Versions](#) on the bucket you will see the different versions of your state file.

**Objects (3)**

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

	Name	Type	Version ID	Last modified	Size	Storage class
<input type="checkbox"/>	<a href="#">aws_infra</a>	-	WBZubSfCWeVmugK.aK_i3XeKoopuhJK		47.8 KB	Standard
<input type="checkbox"/>	<a href="#">aws_infra</a>	-	2BeR91EDXoQLGpTlY6ltPHm3r5b3LtQh		47.8 KB	Standard
<input type="checkbox"/>	<a href="#">aws_infra</a>	-	null		813.0 B	Standard

**Figure 2:** Show Versions

### Step 1.4 - Enable Encryption on S3 Bucket

It is also incredibly important to protect terraform state data as it can contain extremely sensitive information. Store Terraform state in a backend that supports encryption. Instead of storing your state in a local `terraform.tfstate` file. Many backends support encryption, so that instead of your state files being in plain text, they will always be encrypted, both in transit (e.g., via TLS) and on disk (e.g., via AES-256). The `s3` backend supports encryption, which reduces worries about storing sensitive data in state files.





## Edit default encryption Info

### Default encryption

Automatically encrypt new objects stored in this bucket. [Learn more](#)

#### Server-side encryption

- Disable
- Enable

#### Encryption key type

To upload an object with a customer-provided encryption key (SSE-C), use the AWS CLI, AWS SDK, or Amazon S3 REST API.

- Amazon S3 key (SSE-S3)**

An encryption key that Amazon S3 creates, manages, and uses for you. [Learn more](#)

- AWS Key Management Service key (SSE-KMS)**

An encryption key protected by AWS Key Management Service (AWS KMS). [Learn more](#)

[Cancel](#)

[Save changes](#)

**Figure 3:** Enable S3 Encryption

Anyone on your team who has access to that S3 bucket will be able to see the state files in an unencrypted form, so this is still a partial solution, but at least the data will be encrypted at rest (S3 supports server-side encryption using AES-256) and in transit (Terraform uses SSL to read and write data in S3).

### Step 1.4 - Enable Locking for S3 Backend

The `s3` backend stores Terraform state as a given key in a given bucket on Amazon S3 to allow everyone working with a given collection of infrastructure the ability to access the same state data. In order to prevent concurrent modifications which could cause corruption, we need to implement locking on the backend. The `s3` backend supports state locking and consistency checking via Dynamo DB.

State locking for the `s3` backend can be enabled by setting the `dynamodb_table` field to an existing DynamoDB table name. A single DynamoDB table can be used to lock multiple remote state files.

Create a DynamoDB table





Hands-On Labs

Database

## Amazon DynamoDB

### A fast and flexible NoSQL database service for any scale

DynamoDB is a fully managed, key-value, and document database that delivers single-digit-millisecond performance at any scale.

**Get started**  
Create a new table to start exploring DynamoDB.  
[Create table](#)

**Pricing**

**Figure 4:** Create DynamoDB



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



### Create table

#### Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

#### Table name

This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (\_), hyphens (-), and periods (.)

#### Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

String



1 to 255 characters and case sensitive.

#### Sort key - optional

You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

String



1 to 255 characters and case sensitive.

### Settings

#### Default settings

The fastest way to create your table. You can modify these settings now or after your table has been created.

#### Customize settings

Use these advanced features to make DynamoDB work better for your needs.

**Figure 5:** Config DynamoDB

Tables (1) <small>Info</small>							
Actions		Delete		Create table			
<input type="text"/> Find tables by table name		<input type="text" value="Any table tag"/>		< 1 >			
Name	Status	Partition key	Sort key	Indexes	Read capacity mode	Write capacity mode	Encryption
terraform-locks	Active	LockID (String)	-	0	Provisioned with auto scaling (5)	Provisioned with auto scaling (5)	Default

**Figure 6:** DynamoDB Complete

Update the `s3` backend to use the new DynamoDB Table and reconfigure your backend.





```
terraform {  
  backend "s3" {  
    # Replace this with your bucket name!  
    bucket = "myterraformstate"  
    key    = "path/to/my/key"  
    region = "us-east-1"  
  
    # Replace this with your DynamoDB table name!  
    dynamodb_table = "terraform-locks"  
    encrypt        = true  
  }  
}
```

```
terraform init -reconfigure
```

```
Initializing the backend...
```

```
Successfully configured the backend "s3"! Terraform will automatically  
use this backend unless the backend configuration changes.
```

Update the size of your web server from `t2.small` to a `t2.micro` and apply the change.

```
resource "aws_instance" "web_server_2" {  
  ami           = data.aws_ami.ubuntu.id  
  instance_type = "t2.micro"  
  subnet_id     = aws_subnet.public_subnets["public_subnet_2"].id  
  tags = {  
    Name = "Web EC2 Server 2"  
  }  
}
```

```
terraform apply
```

Now you can see that your state file is locked by selecting the DynamoDB table and looking at [View Items](#) to see the lock.





## Hands-On Labs

The screenshot shows the AWS DynamoDB Items page. On the left, there's a sidebar with 'Tables (1)' and a 'terraform-locks' table selected. The main area shows a table with one item:

LockID	Info
my-terraform-state-ghm/prod/aws_infra	{"ID": "c611fbcc-1080-d661-d783-cfb14e56d65", "Operation": "OperationTypeApply", "Info": "", "Who": "gabe@MacBook-Pro..."}

**Figure 7:** View Lock

### Step 1.4 - Remove existing resources with `terraform destroy`

Before exploring other types of state backends we will issue a cleanup of our infrastructure using a `terraform destroy` before changing our our backend in the next steps. This is not a requirement as Terraform supports the migration of state data between backends, which will be covered in a future lab.

```
terraform destroy

Plan: 0 to add, 0 to change, 27 to destroy.

Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes
```

Note: You delete any instances of the `terraform.tfstate` or `terraform.tfstate.backup` items locally as your state is now being managed remotely. Although not required with standard backends it is helpful to keep your working directory clean.





## Task 2: Standard Backend: HTTP Backend (Optional)

Let's take a look at a different standard Terraform backend type - [http](#). A configuration can only provide one backend block, so let's update our configuration to utilize the [http](#) backend instead of [s3](#).

To use this backend we will first need to provide a simple HTTP Server for which Terraform can store its state. This lab is optional because not all students will be able to launch or have access to a HTTP server. This lab will use a simple Simple HTTP server that is being executed via python. The source code of this HTTP server is available for use.

### Step 2.1 - Initiate HTTP Server

Copy the HTTP Server code to a new directory (for example: [webserver](#)) and after following the instructions start the web server using the following command in a terminal:

```
cd webserver  
python -m SimpleHTTPServer 8000
```

```
* Serving Flask app 'stateserver' (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production  
  deployment.  
  Use a production WSGI server instead.  
* Debug mode: on  
* Running on all addresses.  
  WARNING: This is a development server. Do not use it in a production  
  deployment.  
* Running on http://192.168.1.202:5000/ (Press CTRL+C to quit)  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: 158-806-515
```

### Step 2.2 - Update Terraform configuration to use HTTP Standard Backend

Update your Terraform configuration block to use the [http](#) backend pointing to the webserver that was just launched.

```
terraform.tf
```

```
terraform {  
  backend "http" {
```





```

address      = "http://localhost:5000/terraform_state/my_state"
lock_address = "http://localhost:5000/terraform_lock/my_state"
lock_method  = "PUT"
unlock_address = "http://localhost:5000/terraform_lock/my_state"
unlock_method = "DELETE"
}
}

```

Note: A Terraform configuration can only specify a single backend. If a backend is already configured be sure to replace it. Copy just the backend block above and not the full terraform block You can validate the syntax is correct by issuing a `terraform validate`

### Step 2.3 - Re-initialize Terraform and Validate HTTP Backend

```
terraform init -reconfigure
```

```
Initializing the backend...
```

```
Successfully configured the backend "http"! Terraform will automatically
use this backend unless the backend configuration changes.
```

```
terraform apply
```

```
Plan: 27 to add, 0 to change, 0 to destroy.
```

```
Do you want to perform these actions?
```

```
Terraform will perform the actions described above.
```

```
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
...
```

```
Apply complete! Resources: 27 added, 0 changed, 0 destroyed.
```

View the HTTP Logs in the webserver terminal to showcase the use of the `http` backend

```

127.0.0.1 -- [] "GET /terraform_state/my_state HTTP/1.1" 404 -
[,636] DEBUG in stateserver: PUT for http://localhost:5000/terraform_lock/
my_state...
Header -- Host = localhost:5000
Header -- User-Agent = Go-http-client/1.1
Header -- Content-Length = 199
Header -- Content-Md5 = RlUK/ZscZNV/YNHQw0gyNw==
Header -- Content-Type = application/json

```





```

Header -- Accept-Encoding = gzip
Body -- {"ID":"12cae7fb-5f47-d751-8d8d-00ac15b25a00","Operation":"
    "OperationTypeApply","Info":"","Who":"gabe@MacBook-Pro","Version":"
    "1.0.10","Created":"06:47.629496Z","Path":""}

127.0.0.1 - - [] "POST /terraform_state/my_state?ID=12cae7fb-5f47-d751-8
d8d-00ac15b25a00 HTTP/1.1" 200 -
[,998] DEBUG in stateserver: DELETE for http://localhost:5000/
terraform_lock/my_state...
Header -- Host = localhost:5000
Header -- User-Agent = Go-http-client/1.1
Header -- Content-Length = 199
Header -- Content-Md5 = RLUK/ZscZNV/YNHQw0gyNw==
Header -- Content-Type = application/json
Header -- Accept-Encoding = gzip
Body -- {"ID":"12cae7fb-5f47-d751-8d8d-00ac15b25a00","Operation":"
    "OperationTypeApply","Info":"","Who":"gabe@MacBook-Pro","Version":"
    "1.0.10","Created":"06:47.629496Z","Path":""}

```

#### Step 2.4 - View the state, log and lock files.

You can view the remote state file in the `http` backend by going into the webserver directory and looking inside the `.stateserver` directory. Both the state file and the log are located in this directory. This backend also supports state locking creating a `my_state.lock` when a lock is applied.

```

webserver
|--- .stateserver
|   |--- my_state
|   |--- my_state.log
|--- requirements.txt
|--- stateserver.py

```

#### Step 2.5 - Remove existing resources with `terraform destroy`

Before exploring other backends we will issue a cleanup of our infrastructure using a `terraform destroy` before changing our our backend in the next steps. This is not a requirement as Terraform supports the migration of state data between backends, which will be covered in a future lab.

```

terraform destroy

Plan: 0 to add, 0 to change, 27 to destroy.

```





Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only '`yes`' will be accepted to confirm.

Enter a value: `yes`

Note: You delete any instances of the `terraform.tfstate` or `terraform.tfstate.backup` items locally as your state is now being managed remotely. Although not required with standard backends it is helpful to keep your working directory clean.

#### Step 2.6 - Stop Web Server

Once all the cloud resources have been cleaned up you can stop the HTTP Server in the `webserver` directory.





## Terraform Remote State - Enhanced Backend

Enhanced backends can both store state and perform operations. There are only two enhanced backends: `local` and `remote`. The `local` backend is the default backend used by Terraform which we worked with in previous labs. The `remote` backend stores Terraform state and may be used to run operations in Terraform Cloud. When using full remote operations, operations like `terraform plan` or `terraform apply` can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal. Remote plans and applies use variable values from the associated Terraform Cloud workspace.

- Task 1: Log in to Terraform Cloud
- Task 2: Update Terraform configuration to use Remote Enhanced Backend
- Task 3: Re-initialize Terraform and Validate Remote Backend
- Task 4: Provide Secure Credentials for Remote Runs
- Task 5: View the state, log and lock files in Terraform Cloud
- Task 6: Remove existing resources with `terraform destroy`

Let's take a closer look at the `remote` enhanced backend.

### Task 1: Log in to Terraform Cloud

Log in to Terraform Cloud and determine your organization name.

Note: You created a Terraform Cloud Organization in an earlier lab

Open the organization switcher menu in the top navigation bar and take note of your organization name.



A screenshot of the HashiCorp Terraform interface. On the left, there's a sidebar with a dark blue header containing the HashiCorp logo and the word "Organization". Below the header, the sidebar lists three organizations: "Enterprise-Cloud" (which is highlighted in a blue box), "Enterprise-DataCenter", and "example-org-5a4eda". To the right of the sidebar, the main area has a dark blue header with the text "Choose an organization ▾".

Choose an organization ▾

Enterprise-Cloud

Enterprise-DataCenter

example-org-5a4eda

**Figure 1:** Organization Name

## Task 2: Update Terraform configuration to use Remote Enhanced Backend

A configuration can only provide one backend block, so let's update our configuration to utilize the `remote` backend.

`terraform.tf`

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "YOUR-ORGANIZATION"  
  
    workspaces {  
      name = "my-aws-app"  
    }  
  }  
}
```

Update the `organization` name with what you noted in the previous step.





Example:

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "Enterprise-Cloud"  
  
    workspaces {  
      name = "my-aws-app"  
    }  
  }  
}
```

Note: A Terraform configuration can only specify a single backend. If a backend is already configured be sure to replace it. Copy just the backend block above and not the full terraform block. You can validate the syntax is correct by issuing a `terraform validate`

Be sure to issue a `terraform destroy` and delete any instances of the `terraform.tfstate` or `terraform.tfstate.backup` items locally as your state will be managed remotely.

### Task 3: Re-initialize Terraform and Validate Remote Backend

```
terraform init -reconfigure
```

Initializing the backend...

Successfully configured the backend "remote"! Terraform will automatically use `this` backend unless the backend configuration changes.

```
terraform apply
```

Running apply in the remote backend. Output will stream here. Pressing Ctrl-C will cancel the remote apply if it's still pending. If the apply started it will stop streaming the logs, but will not stop the apply running remotely.

Preparing the remote apply...

To view this run in a browser, visit:  
<https://app.terraform.io/app/Enterprise-Cloud/my-aws-app/runs/run-SThbeRjvZUUpH4e9>





Notice that the Terraform apply with the `remote` backend looks different than with the standard backends. The enhanced Terraform `remote` backend stores the state information within Terraform Cloud as well as performs all operations from Terraform Cloud. Therefore this backend supports the ability to centrally store state and centrally manage the Terraform workflow.

This can be validated by the messages returned when executing Terraform CLI commands using the `remote` backend.

#### Task 4: Provide Secure Credentials for Remote Runs

Now that the terraform workflow is being run using the `remote` backend inside Terraform Cloud, we have to configure Terraform Cloud to use our AWS Credentials for building out our infrastructure. In fact during our last step you most likely encountered an error similar to the following:

```
Error: error configuring Terraform AWS Provider: no valid credential sources for Terraform AWS Provider found.

Please see https://registry.terraform.io/providers/hashicorp/aws
for more information about providing credentials.
```

We can provide Terraform Cloud with our AWS Credentials as environment variables inside the `Variables` section of our workspace.

Enterprise-Cloud / Workspaces / my-aws-app / Variables

my-aws-app	Resources 0	Terraform version 1.0.10	Updated 3 hours ago
No workspace description available. <a href="#">Add workspace description</a> .			
<a href="#">Overview</a> <a href="#">Runs</a> <a href="#">States</a> <b>Variables</b> <a href="#">Settings</a> <span style="float: right;"><a href="#">Actions</a> <span style="font-size: small;">Unlocked</span> </span>			

**Variables**

Terraform uses all [Terraform](#) and [Environment](#) variables for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration. You may want to use the Terraform Cloud Provider or the variables API to add multiple variables at once.

**Sensitive variables**

[Sensitive](#) variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

**Workspace variables (0)**

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence [»](#).

Key	Value	Category
There are no variables added.		
<a href="#">+ Add variable</a>		

**Figure 2:** Workspace Variables





## Hands-On Labs

Select [Add variable](#), Select [Environment variable](#). Create both a `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variable with the AWS credentials. Now whenever the Terraform workflow is executed using our configured `remote` backend, Terraform Cloud knows which credentials to use to access our AWS cloud account.

You can mark one or both of these variables as [Sensitive](#) so that others cannot view their values.

[my-aws-app](#)

No workspace description available. [Add workspace description](#).

Overview	Runs	States	Variables	Settings	Resources	Terraform version	Updated
					0	1.0.10	3 hours ago
						Unlocked	<a href="#">Actions</a>

### Variables

Terraform uses all [Terraform](#) and [Environment](#) variables for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration. You may want to use the Terraform Cloud Provider or the variables API to add multiple variables at once.

#### Sensitive variables

[Sensitive](#) variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

#### Workspace variables (2)

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence [»](#).

Key	Value	Category	...
<code>AWS_ACCESS_KEY_ID</code>	AKIAJAXMT4EY5DLVHLRJH6	env	...
<code>AWS_SECRET_ACCESS_KEY</code> SENSITIVE	Sensitive - write only	env	...

[+ Add variable](#)

**Figure 3:** TFC AWS Variables

Once set you can build out the infrastructure

```
terraform apply

Running apply in the remote backend. Output will stream here. Pressing
Ctrl-C
will cancel the remote apply if it's still pending. If the apply started
it
will stop streaming the logs, but will not stop the apply running remotely
.

Preparing the remote apply...

To view this run in a browser, visit:
https://app.terraform.io/app/Enterprise-Cloud/my-aws-app/runs/run-2
uxkrzPJ62pmHrQ6

Waiting for the plan to start...
```



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



```
Terraform v1.0.10  
on linux_amd64  
Configuring remote state backend...  
Initializing Terraform configuration...  
....  
Plan: 27 to add, 0 to change, 0 to destroy.  
  
Do you want to perform these actions in workspace "my-aws-app"?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.  
  
Enter a value: yes
```

You can view the apply also within the Terraform Cloud UI.

A screenshot of the Terraform Cloud workspace interface. At the top, it shows a workspace named "my-aws-app". Below the workspace name, there's a message stating "No workspace description available. [Add](#) workspace description." Underneath this, there are tabs for "Overview", "Runs" (which is selected), "States", "Variables", and "Settings". On the right side, there are status indicators for "Resources" (0), "Terraform version" (1.0.10), and "Updated" (a few seconds ago). Below these, there's a "Actions" button. The main content area is titled "Current Run" and shows a single run entry. This entry includes a user icon, the trigger information "Triggered via CLI [CURRENT] #run-2uxkrzPJ62pmHrQ6 | gabe\_maentz triggered via CLI", and a status indicator "Applying" with a timestamp "a few seconds ago". Below this, there's a "Run List" section which shows the same run entry again, indicating it is the current run.

**Figure 4:** Terraform Cloud Apply



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

my-aws-app

No workspace description available. [Add workspace description](#).

Resources 0      Terraform version 1.0.10      Updated a few seconds ago

Overview    **Runs**    States    Variables    Settings    Actions

**Triggered via CLI**

**CURRENT**

gabe\_maentz triggered a run from CLI 2 minutes ago

Plan finished 2 minutes ago      Resources: 27 to add, 0 to change, 0 to destroy

Apply running a minute ago      Started a minute ago

View raw log

```
aws_instance.ubuntu_server: Creating...
aws_subnet_public_subnets["public_subnet_2"]: Creation complete after 12s [id=subnet-05205168af0a8552f]
aws_route_table_association.public["public_subnet_2"]": Creating...
aws_nat_gateway.nat_gateway: Creating...
aws_route_table_association.public["public_subnet_1"]": Creating...
aws_route_table_association.public["public_subnet_3"]": Creating...
aws_instance.web_server_2: Creating...
aws_route_table_association.public["public_subnet_1"]": Creation complete after 0s [id=rtbassoc-0ef85fa24f466c773]
aws_route_table_association.public["public_subnet_3"]": Creation complete after 1s [id=rtbassoc-0d6d4ba8757430ace]
aws_instance.ubuntu_server: Still creating... [10s elapsed]
aws_route_table_association.public["public_subnet_1"]": Still creating... [10s elapsed]
aws_instance.web_server_2: Still creating... [10s elapsed]
aws_instance.ubuntu_server: Still creating... [20s elapsed]
aws_instance.web_server_2: Still creating... [20s elapsed]
aws_instance.ubuntu_server: Still creating... [30s elapsed]
aws_instance.web_server_2: Still creating... [30s elapsed]
aws_instance.ubuntu_server: Still creating... [40s elapsed]
aws_instance.web_server_2: Still creating... [40s elapsed]
aws_instance.ubuntu_server: Still creating... [42s elapsed]
aws_instance.web_server_2: Creation complete after 42s [id=i-0dba0a3c2a5be1bf6]
aws_instance.ubuntu_server: Still creating... [50s elapsed]
```

**Figure 5:** Terraform Cloud Apply Detail

### Task 5: View the state, log and lock files in Terraform Cloud

Using the `remote` backend allows you to connect to Terraform Cloud. Within the Terraform Cloud UI you'll be able to:

- View Terraform run history
- View state history
- View all your organization's workspaces
- Lock a workspace, making it easy to avoid conflicting changes and state corruption
- Execute the Terraform workflow via CLI or UI

Let's view the state file in Terraform Cloud.

- If you aren't already, log in to Terraform Cloud in your browser
- Navigate to your Workstation
- In your Workstation UI page, click on the [States](#) tab





## Hands-On Labs

- You should see something along the lines of this in the [States](#) tab. This indicates that your state file is now being stored and versioned within Terraform Cloud using the [remote](#) backend.

my-aws-app

No workspace description available. [Add workspace description](#).

Overview Runs States Variables Settings Actions

Resources 30 Terraform version 1.0.10 Updated a few seconds ago

Triggered via CLI #sv-2YVbichjpvDVND2w | gabe\_maentz triggered from Terraform | #run-2uxkrzPJ62pmHrQ6

Download 3 minutes ago

filter Apply Learn more about filtering JSON data. Expand Full screen

```
1 {
2   "version": 4,
3   "terraform_version": "1.0.10",
4   "serial": 0,
5   "lineage": "ae54c6de-88ca-568c-39af-68b6f4c8511f",
6   "outputs": {},
7   "resources": [
8     {
9       "mode": "data",
10      "type": "aws_ami",
11      "name": "ubuntu",
12      "provider": "provider{\"registry.terraform.io/hashicorp/aws\":}",
13      "instances": [
14        {
15          "schema_version": 0,
16          "attributes": {
17            "architecture": "x86_64",
18          }
19        }
20      ]
21    }
22  ]
23}
```

## Task 6: Remove existing resources with `terraform destroy`

We will now issue a cleanup of our infrastructure using a [terraform destroy](#)

```
terraform destroy
```

```
Plan: 0 to add, 0 to change, 27 to destroy.
```

```
Do you really want to destroy all resources?
```

```
Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only 'yes' will be accepted to confirm.
```

```
Enter a value: yes
```





## Lab: Terraform State Migration

As your maturity and use of Terraform develops there may come a time when you need change the backend type you are using. Perhaps you are onboarding new employees and now need to centralize state. You might be part of a merger/acquisition where you need to onboard another organization's Terraform code to your standard configuration. You may simply like to move from a standard backend to an enhanced backend to leverage some of those backend features. Luckily Terraform makes it relatively easy to change your state backend configuration and migrate the state between backends along with all of the data that the state file contains.

- Task 1: Use Terraform's default `local` backend
- Task 2: Migrate State to `s3` backend
- Task 3: Migrate State to `remote` backend
- Task 4: Migrate back to `local` backend

### Task 1: Use Terraform's default `local` backend

Update the terraform configuration block within the `terraform.tf` and remove the `backend`. This will indicate to Terraform to use its default `local` backend and store the contents of state inside a `terraform.tfstate` file locally inside the working directory.

```
terraform {  
    required_version = ">= 1.0.0"  
    required_providers {  
        aws = {  
            source  = "hashicorp/aws"  
            version = "~> 3.0"  
        }  
        http = {  
            source  = "hashicorp/http"  
            version = "2.1.0"  
        }  
        random = {  
            source  = "hashicorp/random"  
            version = "3.1.0"  
        }  
        local = {  
            source  = "hashicorp/local"  
            version = "2.1.0"  
        }  
        tls = {  
            source  = "hashicorp/tls"  
        }  
    }  
}
```





```
        version = "3.1.0"
    }
}
```

Validate your configuration and re-initialize to terraform's default `local` backend.

```
terraform validate
terraform init -migrate-state
```

Initializing the backend...

Terraform has detected you're unconfiguring your previously set "remote" backend.

Successfully unset the backend "remote". Terraform will now operate locally.

Initializing provider plugins...

- Reusing previous version of hashicorp/tls from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/http from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Using previously-installed hashicorp/tls v3.1.0
- Using previously-installed hashicorp/aws v3.65.0
- Using previously-installed hashicorp/http v2.1.0
- Using previously-installed hashicorp/random v3.1.0
- Using previously-installed hashicorp/local v2.1.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

Build the infrastructure and state using a `terraform apply`

```
terraform apply
```





```
Plan: 27 to add, 0 to change, 0 to destroy.
```

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

```
Enter a value: yes
```

Validate the buildout of infrastructure and state were successful by listing the items in state:

```
terraform state list
```

## Task 2: Migrate State to s3 backend

We will now migrate existing state from the default `local` backend to our `s3` backend to store the state information centrally in S3. Update the terraform configuration block within the `terraform.tf` and add our `s3` backend configuration. Remember that a Terraform configuration can only specify a single backend for a given working directory. If a backend is already configured be sure to replace it.

```
terraform.tf
```

Note: Don't forget to update the configuration block below to specify your bucket name, key and DynamoDB table name that were created in earlier labs.

Example:

```
terraform {  
  backend "s3" {  
    bucket = "my-terraform-state-ghm"  
    key    = "prod/aws_infra"  
    region = "us-east-1"  
  
    dynamodb_table = "terraform-locks"  
    encrypt        = true  
  }  
}
```

Validate your configuration and re-initialize to terraform's `s3` backend.

```
terraform validate  
terraform init -migrate-state
```

```
Success! The configuration is valid.
```





Initializing the backend...

Do you want to copy existing state to the **new** backend?

Pre-existing state was found **while** migrating the previous "**local**" backend to the

newly configured "**s3**" backend. No existing state was found in the newly configured "**s3**" backend. Do you want to copy **this** state to the **new** "**s3**" backend? Enter "**yes**" to copy and "**no**" to start with an empty state.

Enter a value: yes

Successfully configured the backend "**s3**"! Terraform will automatically use **this** backend unless the backend configuration changes.

Validate that the migration was successful by listing the items in state using the new backend.

```
terraform state list
```

Properties	Permissions	Versions
Owner: gabe AWS Region: US East (N. Virginia) us-east-1 Last modified: [timestamp] Size: 47.1 KB Type: [file type] Key: prod/aws_infra	S3 URI: s3://my-terraform-state-ghm/prod/aws_infra Amazon Resource Name (ARN): arn:aws:s3:::my-terraform-state-ghm/prod/aws_infra Entity tag (Etag): 23ee66ae9e08fbcc1e8720ace087df0 Object URL: https://my-terraform-state-ghm.s3.amazonaws.com/prod/aws_infra	

**Figure 1:** Remote State S3

Congratulations! You have successfully migrated active state from the **local** to **s3** backend and are now storing state remotely. With the **s3** standard backend you are able to share your workspace with teammates, leverage state locking, versioning and the encryption features of the **s3** backend.

If you can read the state information then the migration was successful. You can now remove the local `terraform.tfstate` and `terraform.tfstate.backup` files locally in your working directory.





### Task 3: Migrate State to remote backend

We will now migrate existing state from the `s3` standard backend to the `remote` enhanced backend to store the state information centrally in Terraform Cloud. Update the terraform configuration block within the `terraform.tf` to replace the `s3` backend configuration with the `remote` backend configuration. Remember that a Terraform configuration can only specify a single backend for a given working directory.

`terraform.tf`

Note: Don't forget to update the configuration block below to specify your Terraform Cloud organization and workspace name that were created in earlier labs.

Example:

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "Enterprise-Cloud"  
  
    workspaces {  
      name = "my-aws-app"  
    }  
  }  
}
```

Validate your configuration and re-initialize to terraform's `remote` backend.

```
terraform validate  
terraform init -migrate-state
```

```
Initializing the backend...  
Backend configuration changed!
```

Terraform has detected that the configuration specified `for` the backend has changed. Terraform will now check `for` existing state in the backends.

```
Terraform detected that the backend type changed from "s3" to "remote".  
Do you want to copy existing state to the new backend?  
Pre-existing state was found while migrating the previous "s3" backend  
to the  
newly configured "remote" backend. No existing state was found in the  
newly  
configured "remote" backend. Do you want to copy this state to the new "  
remote"
```





backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value: yes

Successfully configured the backend "remote"! Terraform will automatically use **this** backend unless the backend configuration changes.

Validate that the migration was successful by listing the items in state using the new backend.

`terraform state list`

Enterprise-Cloud / Workspaces / my-aws-app / States / sv-HXTpLqzW3gxBX43o

my-aws-app	Resources	Terraform version	Updated
No workspace description available. <a href="#">Add workspace description.</a>	30	1.0.10	a few seconds ago

Overview    Runs    **States**    Variables    Settings    Actions

New state #sv-HXTpLqzW3gxBX43o  
gabe\_maentz triggered from Terraform

Download    a minute ago

```

filter
Apply
① Learn more about filtering JSON data.
Expand
Full screen
1
2 "version": 4,
3 "terraform_version": "1.0.10",
4 "serial": 36,
5 "lineage": "6fa51cc8-46f9-c074-a917-7415f976bf44",
6 "outputs": {},
7 "resources": [
8   {
9     "mode": "data",
10    "type": "aws_ami",
11    "name": "ubuntu",
12    "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
13    "instances": [
14      {
15        "schema_version": 0,
16        "attributes": {
17          "architecture": "x86_64",
18

```

**Figure 2:** Remote State TFC

Congratulations! You have successfully migrated active state from the `s3` to `remote` backend and are now storing state remotely in Terraform Cloud. With the `remote` enhanced backend you are able to share your workspace with teammates, leverage state locking, versioning, encryption and centrally perform operations using the features of Terraform Cloud.

#### Task 4: Migrate back to `local` backend

Now that we have migrated our state to several different backend types, let's show how to restore the Terraform state back to its default `local` backend.





## Hands-On Labs

Update the terraform configuration block within the `terraform.tf` and remove the `backend`. This will indicate to Terraform to use it's default `local` backend and store the contents of state inside a `terraform.tfstate` file locally inside the working directory.

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "3.1.0"
    }
  }
}
```

Validate your configuration and re-initialize to terraform's `local` backend.

```
terraform validate
terraform init -migrate-state
```

```
Initializing the backend...
Terraform has detected you're unconfiguring your previously set "remote" backend.
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "remote" backend to the newly configured "local" backend. No existing state was found in the newly configured "local" backend. Do you want to copy this state to the new "local" backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value: yes
```





## Hands-On Labs

Successfully unset the backend "remote". Terraform will now operate locally.

Validate that the migration was successful by listing the items in state using the new backend.

```
terraform state list
```

As part of this migration back to the `local` backend you will see that terraform created a `terraform.tfstate` file in which the state is now being managed.

Congratulations! You have successfully migrated active state from `local` to `s3` to `remote` and back again. You are once again storing state locally within a `terraform.tfstate` file of your working directory. As you can see, Terraform makes it relatively easy to change your state backend configuration and migrate to the backend that is appropriate for your team.





## Lab: Terraform State Refresh

The `terraform refresh` command reads the current settings from all managed remote objects found in Terraform state and updates the Terraform state to match.

Refreshing state is the first step of the `terraform plan` process: read the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date. If you wish to only perform the first part of the `terraform plan` process you can execute the plan with a `-refresh-only`.

- Task 1: `terraform refresh` command
- Task 2: Show state before refresh
- Task 3: Introduce drift
- Task 4: Refresh state to detect drift
- Task 5: Show state after refresh
- Task 6: Controlling state refresh

### Task 1: `terraform refresh` command

The `terraform refresh` command won't modify your real remote objects, but it will modify the the Terraform state. You shouldn't typically need to use this command, because Terraform automatically performs the same refreshing actions as a part of creating a plan in both the `terraform plan` and `terraform apply` commands. Once your infrastructure is deployed you can run a `terraform refresh` to read the current settings of objects that terraform is managing.

Deploy your infrastructure and run a `terraform refresh`

```
terraform init  
terraform apply  
terraform refresh
```

### Task 2: Show State before Refresh

Use the `terraform state` commands to show the state of an EC2 instance before a state refresh is applied.

```
terraform state list  
  
data.aws_ami.ubuntu  
data.aws_availability_zones.available  
data.aws_region.current
```





```

aws_eip.nat_gateway_eip
aws_instance.ubuntu_server
aws_instance.web_server
aws_instance.web_server_2
aws_internet_gateway.internet_gateway
aws_key_pair.generated
aws_nat_gateway.nat_gateway
aws_route_table.private_route_table
aws_route_table.public_route_table
aws_route_table_association.private["private_subnet_1"]
aws_route_table_association.private["private_subnet_2"]
aws_route_table_association.private["private_subnet_3"]
aws_route_table_association.public["public_subnet_1"]
aws_route_table_association.public["public_subnet_2"]
aws_route_table_association.public["public_subnet_3"]
aws_security_group.ingress-ssh
aws_security_group.vpc-ping
aws_security_group.vpc-web
aws_subnet.private_subnets["private_subnet_1"]
aws_subnet.private_subnets["private_subnet_2"]
aws_subnet.private_subnets["private_subnet_3"]
aws_subnet.public_subnets["public_subnet_1"]
aws_subnet.public_subnets["public_subnet_2"]
aws_subnet.public_subnets["public_subnet_3"]
aws_vpc.vpc
local_file.private_key_pem
random_string.random
tls_private_key.generated
module.server.aws_instance.web
module.server_subnet_1.aws_instance.web

```

```
terraform state show aws_instance.web_server
```

```

resource "aws_instance" "web_server" {
  ami                               = "ami-04505e74c0741db8d"
  arn                               = "arn:aws:ec2:us-east
  -1:508140242758:instance/i-0fdab85c96f92f0e9"
  associate_public_ip_address       = true
  availability_zone                 = "us-east-1b"
  cpu_core_count                   = 1
  cpu_threads_per_core             = 1
  disable_api_termination          = false
  ebs_optimized                    = false
  get_password_data                = false
  hibernation                      = false
  id                               = "i-0fdab85c96f92f0e9"
  instance_initiated_shutdown_behavior = "stop"
  instance_state                   = "running"
}

```





## Hands-On Labs

```

instance_type = "t2.micro"
ipv6_address_count = 0
ipv6_addresses = []
key_name = "MyAWSKey"
monitoring = false
primary_network_interface_id = "eni-02b1bae8ab8850b6c"
private_dns = "ip-10-0-101-122.ec2.internal"
private_ip = "10.0.101.122"
public_dns = "ec2-3-236-76-73.compute-1.amazonaws.com"
public_ip = "3.236.76.73"
secondary_private_ips = []
security_groups = [
  "sg-0609dcff9a0ad9607",
  "sg-06b21df8181c04026",
  "sg-0b3632ec4e1686072",
]
source_dest_check = true
subnet_id = "subnet-0422128dd82c11546"
tags = {
  "Name" = "Web EC2 Server"
}
tags_all = {
  "Name" = "Web EC2 Server"
}
tenancy = "default"
vpc_security_group_ids = [
  "sg-0609dcff9a0ad9607",
  "sg-06b21df8181c04026",
  "sg-0b3632ec4e1686072",
]

capacity_reservation_specification {
  capacity_reservation_preference = "open"
}

credit_specification {
  cpu_credits = "standard"
}

enclave_options {
  enabled = false
}

metadata_options {
  http_endpoint = "enabled"
  http_put_response_hop_limit = 1
  http_tokens = "optional"
}

```





```
root_block_device {  
    delete_on_termination = true  
    device_name          = "/dev/sda1"  
    encrypted            = false  
    iops                 = 100  
    tags                 = []  
    throughput           = 0  
    volume_id            = "vol-012dff0c9b3ce206e"  
    volume_size          = 8  
    volume_type          = "gp2"  
}  
}
```

### Task 3: Introduce Drift

Change the tag of a given object inside the AWS Console

The screenshot shows the 'Manage tags' dialog for an EC2 instance. The URL in the browser bar is 'EC2 > Instances > i-0fdab85c96f92f0e9 > Manage tags'. The dialog title is 'Manage tags' with an 'Info' link. A descriptive text states: 'A tag is a custom label that you assign to an AWS resource. You can use tags to help organize and identify your instances.' Below this, there is a table with two columns: 'Key' and 'Value - optional'. In the 'Key' column, there is a search input field with 'Name' and a clear button 'X'. In the 'Value' column, there is a search input field with 'Web EC2 Server - My App' and a clear button 'X'. To the right of the value input is a 'Remove' button. Below the table is a 'Cancel' button and an orange 'Save' button. At the bottom left of the dialog, it says 'You can add 49 more tags.'

**Figure 1:** AWS Tag - Drift

### Task 4: Refresh State to detect drift

Run a `terraform refresh` command to update Terraform state

```
terraform refresh
```





## Task 5: Show State after Refresh

Use the `terraform state` commands to show the state of an EC2 instance after a state refresh is applied.

```
terraform state show aws_instance.web_server

tags = {
  "Name" = "Web EC2 Server - My App"
}
```

You will see that object within terraform state has been updated to reflect the drift introduced by applying changes outside of the standard Terraform workflow. If these changes are not updated within the Terraform configuration the drift will be highlighted during the next `terraform plan` and will be reverted the next time a `terraform apply` is performed.

```
terraform plan
```

Terraform will perform the following actions:

```
# aws_instance.web_server will be updated in-place
~ resource "aws_instance" "web_server" {
    id                      = "i-0fdab85c96f92f0e9"
    ~ tags                  = {
        ~ "Name" = "Web EC2 Server - My App" -> "Web EC2 Server"
    }
    ~ tags_all              = {
        ~ "Name" = "Web EC2 Server - My App" -> "Web EC2 Server"
    }
    # (28 unchanged attributes hidden)

    # (5 unchanged blocks hidden)
}

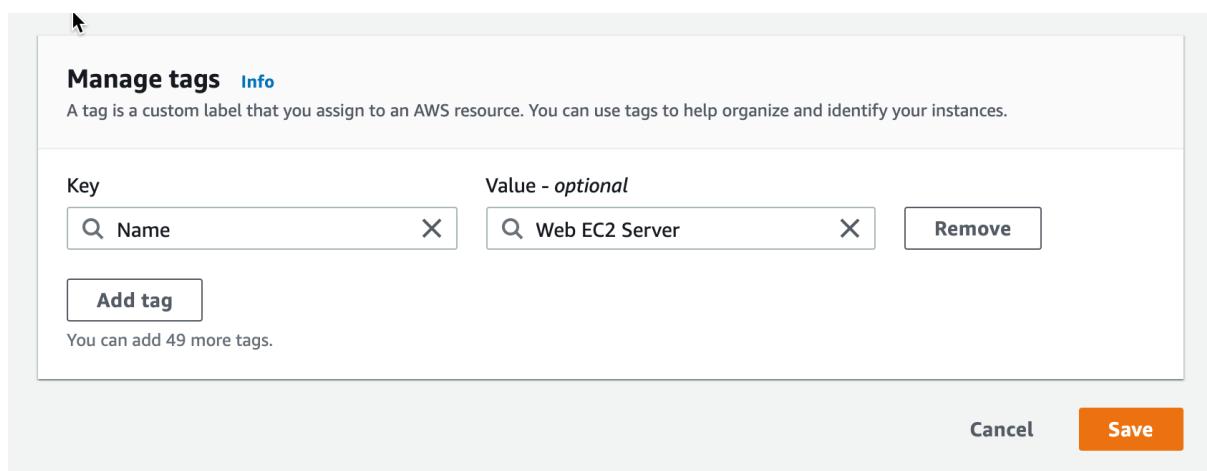
Plan: 0 to add, 1 to change, 0 to destroy.
```

```
terraform apply
```

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

```
Enter a value: yes
```





**Figure 2:** AWS Tag - Revert

You can see the out of band changes were reverted to remediate the drift introduced outside of the standard Terraform workflow. Alternatively, you could keep these changes but would need to update the Terraform configuration appropriately.

### Task 6: Controlling state refresh

Automatically applying the effect of a refresh is risky, because if you have misconfigured credentials for one or more providers then the provider may be misled into thinking that all of the managed objects have been deleted, and thus remove all of the tracked objects without any confirmation prompt. This is why the `terraform refresh` command has been deprecated.

Instead it is recommended to use the `-refresh-only` option to get the same effect as a `terraform refresh` but with the opportunity to review the changes that Terraform has detected before committing them to the state. The first step of the `terraform plan` process is to read the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date. It will then compare the current configuration to the prior state and note differences. If you wish to only perform the first part of this flow you can execute the plan with a `-refresh-only` option to make the.

Change the tag of a given object inside the AWS Console





EC2 > Instances > i-0fdab85c96f92f0e9 > Manage tags

### Manage tags Info

A tag is a custom label that you assign to an AWS resource. You can use tags to help organize and identify your instances.

Key	Value - optional	Remove
<input type="text" value="Name"/> <span>X</span>	<input type="text" value="Web EC2 Server - My App"/> <span>X</span>	<span>Remove</span>
<span>Add tag</span>		

You can add 49 more tags.

Cancel Save

**Figure 3:** AWS Tag - Drift

```
terraform plan -refresh-only

...
Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since
the last "terraform apply":

# aws_instance.web_server has changed
~ resource "aws_instance" "web_server" {
    id                               = "i-0fdab85c96f92f0e9"
    ~ tags                           = {
        ~ "Name" = "Web EC2 Server" -> "Web EC2 Server - My App"
    }
    ~ tags_all                      = {
        ~ "Name" = "Web EC2 Server" -> "Web EC2 Server - My App"
    }
    # (28 unchanged attributes hidden)

}
# (5 unchanged blocks hidden)

This is a refresh-only plan, so Terraform will not take any actions to
```





## Hands-On Labs

undo these. If you were expecting these changes then you can apply **this** plan to record the updated values in the Terraform state without changing any remote objects.

Terraform's Refresh-only mode goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform. This can be useful if you've intentionally changed one or more remote objects outside of the usual workflow (e.g. while responding to an incident) and you now need to reconcile Terraform's records with those changes.

To update Terraform's state with these items that were modified outside of the usual workflow run a `terraform apply -refresh-only`

```
terraform apply -refresh-only
```

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the last "`terraform apply`":

```
# aws_instance.web_server has changed
~ resource "aws_instance" "web_server" {
    id                               = "i-0fdab85c96f92f0e9"
    ~ tags                           = {
        ~ "Name" = "Web EC2 Server" -> "Web EC2 Server - My App"
    }
    ~ tags_all                      = {
        ~ "Name" = "Web EC2 Server" -> "Web EC2 Server - My App"
    }
    # (28 unchanged attributes hidden)

}
# (5 unchanged blocks hidden)
```

This is a refresh-only plan, so Terraform will not take any actions to undo these. If you were expecting these changes then you can apply **this** plan to record the updated values in the Terraform state without changing any remote objects.

Would you like to update the Terraform state to reflect these detected changes?





Terraform will write these changes to the state without modifying any real infrastructure.

There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

Use the `terraform state` commands to show the state of an EC2 instance after a state refresh is applied.

```
terraform state show aws_instance.web_server
```

```
tags = {
  "Name" = "Web EC2 Server - My App"
}
```

If this is permanent change then you would need to update the object within Terraform's configuration to reflect this change otherwise it will be reverted back during the next apply. The `terraform refresh` and `-refresh-only` are useful for being able to help detect and handle drift within an environment.

Run a `terraform apply` to eliminate the drift within the environment.

```
terraform apply
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
 ~ update `in-place`

Terraform will perform the following actions:

```
# aws_instance.web_server will be updated in-place
~ resource "aws_instance" "web_server" {
    id = "i-0fdab85c96f92f0e9"
    ~ tags = {
      "Name" = "Web EC2 Server - My App" -> "Web EC2 Server"
    }
    ~ tags_all = {
      "Name" = "Web EC2 Server - My App" -> "Web EC2 Server"
    }
  # (28 unchanged attributes hidden)
```



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



```
# (5 unchanged blocks hidden)
}

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

aws_instance.web_server: Modifying... [id=i-0fdab85c96f92f0e9]
aws_instance.web_server: Modifications complete after 2s [id=i-0
    fdab85c96f92f0e9]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Manage tags Info

A tag is a custom label that you assign to an AWS resource. You can use tags to help organize and identify your instances.

Key	Value - optional	Remove
<input type="text" value="Name"/>	<input type="text" value="Web EC2 Server"/>	<input type="button" value="Remove"/>

You can add 49 more tags.

**Figure 4:** AWS Tag - Eliminate Drift





## Lab: Terraform Backend Configuration

As we have seen the Terraform backend is configured for a given working directory within a terraform configuration file. A configuration can only provide one backend block per working directory. You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable if some arguments are provided automatically by an automation script running Terraform. When some or all of the arguments are omitted, we call this a partial configuration.

- Task 1: Terraform backend block
- Task 2: Partial backend configuration via a file
- Task 3: Partial backend configuration via command line
- Task 4: Declare backend configuration via interactive prompt
- Task 5: Specifying multiple partial backend configurations
- Task 6: Backend configuration from multiple locations
- Task 7: Change state backend configuration back to default

### Task 1: Terraform backend block

By default there is no `backend` configuration block within Terraform configuration. Because no `backend` is included in the configuration Terraform will use its default backend - `local`. This is why we see the `terraform.tfstate` file in our working directory. If we want to be explicit about which backend Terraform should use it can be specified within the `backend` of the terraform configuration block as shown in previous labs.

Update the `terraform.tf` file to specify a local backend configuration with a path to the `terraform.tfstate` file.

```
terraform {  
  backend "local" {  
    path = "terraform.tfstate"  
  }  
}
```

```
terraform init  
terraform plan  
terraform apply
```





## Task 2: Partial backend configuration via a file

Terraform backends do not support interpolations within its configuration block. This means it is not allowed to have variables as values when defining the backend configuration. Terraform does support the ability to specify partial backend configuration that can be merged into the backend block. This partial configuration can be provided via a file and then specified during the `terraform init`.

To specify a file, use the `-backend-config=PATH` option when running `terraform init`. If the file contains secrets it may be kept in a secure data store and downloaded to the local disk before running Terraform.

Create a two new directories. One called `state_configuration` containing two new files called `dev_local.hcl` and `test_local.hcl` and the other called `state_data` which will remain empty for the moment.

`dev_local.hcl`

```
path = "state_data/terraform.dev.tfstate"
```

`test_local.hcl`

```
path = "state_data/terraform.test.tfstate"
```

Below is a sample of the directory structure once these steps are complete.

```
|--- main.tf
|--- modules
|--- state_configuration
|   |--- dev_local.hcl
|   |--- test_local.hcl
|--- state_data
|--- terraform.tf
|--- variables.tf
```

Specify a particular state file for development by using the `backend-config` path:

```
terraform init -backend-config=state_configuration/dev_local.hcl -migrate-state
terraform plan
terraform apply
```

Notice that a new state file that will be created called: `terraform.dev.tfstate`. Cancel out of the apply, and show how you can do the same for your test environment configuration.





Note: If you make a state file configuration change, you most likely will need to provide the `-migrate-state` option to migrate state from one state file to the other. If that is not the intended action, then you will need to use `-reconfigure`. When modifying any terraform backend configuration it is important to understand the intent of your actions and the appropriate commands for that intent.

```
terraform init -backend-config=state_configuration/test_local.hcl -migrate  
-state  
terraform plan  
terraform apply
```

### Task 3: Partial backend configuration via command line

Key/value pairs for a terraform backend configuration can be specified via the init command line. To specify a single key/value pair, use the `-backend-config="KEY=VALUE"` option when running terraform init.

```
terraform init -backend-config="path=state_data/terraform.prod.tfstate" -  
migrate-state  
terraform plan  
terraform apply
```

Note that many shells retain command-line flags in a history file, so this isn't recommended for secrets.

### Task 4: Specifying multiple partial backend configurations

Terraform backend configuration can also be split up into multiple files. If we update our `terraform.tf` to use the Terraform S3 backend we can break out each of the configuration items into a separate configuration files if desired.

#### 4.1 Full configuration within the `terraform.tf`

Update our Terraform configuration to utilize the s3 backend for storing state.

`terraform.tf`

```
terraform {  
  backend "s3" {
```





```

    bucket = "my-terraform-state-ghm"
    key    = "dev/aws_infra"
    region = "us-east-1"
}
}

```

```

terraform init -migrate-state
terraform plan
terraform apply

```

## 4.2 Partial configuration with a `terraform.tf` and `dev-s3-state.hcl`

The “s3” backend supports partial configuration that allows Terraform to be initialized with a dynamically set backend configuration. Let’s update our Terraform configuration block to specify the bare minimum for our s3 backend configuration and utilize partial configuration files to provide the configuration values.

`terraform.tf`

```

terraform {
  backend "s3" {
  }
}

```

Create a `dev-s3-state.hcl` to specify where the state should be saved.

```

bucket = "my-terraform-state-ghm"
key    = "dev/aws_infra"
region = "us-east-1"

```

```

terraform init -backend-config="state_configuration/dev-s3-state.hcl" -
  migrate-state
terraform plan
terraform apply

```

## 4.3 Partial configuration with a `terraform.tf`, `s3-state-bucket.hcl` and `dev-s3-state-key.hcl`

We can break out our backend into multiple configuration files all containing partial configuration

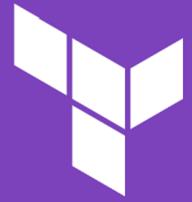
`terraform.tf`

```

terraform {

```





```
backend "s3" {  
}  
}
```

s3-state-bucket.hcl

```
bucket = "my-terraform-state-ghm"  
region = "us-east-1"
```

dev-s3-state-key.hcl

```
key      = "dev/aws_infra"
```

```
terraform init -backend-config="state_configuration/s3-state-bucket.hcl" \  
-backend-config="state_configuration/dev-s3-state-key.hcl" \  
-migrate-state  
terraform plan  
terraform apply
```

### Task 5: Partial backend configuration via CLI prompt

If a required value for a backend configuration item is not specified, Terraform will interactively ask you for the required values, unless interactive input is disabled. Terraform will not prompt for optional values.

Initialize terraform but intentionally leave out the `key` value which is required as part of the s3 backend.

```
terraform init -backend-config="state_configuration/s3-state-bucket.hcl" \  
-migrate-state  
  
Initializing modules...  
  
Initializing the backend...  
Backend configuration changed!  
  
Terraform has detected that the configuration specified for the backend  
has changed. Terraform will now check for existing state in the backends.  
  
key  
The path to the state file inside the bucket  
  
Enter a value: dev/aws_infra
```





## Task 6: Backend configuration from multiple locations

If backend settings are provided in multiple locations, the top-level settings are merged such that any command-line options override the settings in the main configuration and then the command-line options are processed in order, with later options overriding values set by earlier options.

Update the `terraform.tf` configuration to supply all information for the s3 backend as follows:

`terraform.tfstate`

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state-ghm"
    key    = "dev/aws_infra"
    region = "us-east-1"
  }
}
```

Create a `prod-s3-state-key.hcl` file with the `state_configuration` directory with the following partial configuration:

`prod-s3-state-key.hcl`

```
key      = "prod/aws_infra"
```

```
terraform init -backend-config=state_configuration/s3-state-bucket.hcl \
-backend-config=state_configuration/prod-s3-state-key.hcl \
-migrate-state
```

Initializing modules...

Initializing the backend...

Backend configuration changed!

Terraform has detected that the configuration specified **for** the backend has changed. Terraform will now check **for** existing state in the backends.

Do you want to copy existing state to the **new** backend?

Pre-existing state was found **while** migrating the previous "**s3**" backend to the newly configured "**s3**" backend. An existing non-empty state already exists in the **new** backend. The two states have been saved to temporary files that will be removed after responding to **this** query.





```
Previous (type "s3"): /var/folders/1c/qvs1hwp964z_dwd5qg07lv_00000gn/T/
terraform2651199725/1-s3.tfstate
New      (type "s3"): /var/folders/1c/qvs1hwp964z_dwd5qg07lv_00000gn/T/
terraform2651199725/2-s3.tfstate
```

Do you want to overwrite the state in the **new** backend with the previous state?

Enter **"yes"** to copy and **"no"** to start with the existing state in the newly configured **"s3"** backend.

Enter a value: yes

Amazon S3 > my-terraform-state-ghm

**my-terraform-state-ghm** [Info](#)

Objects Properties Permissions Metrics Management Access Points

**Objects (2)**

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

C Copy S3 URI Copy URL Download Open Delete Actions Create folder

Upload

Find objects by prefix Show versions

Name	Type	Last modified	Size	Storage class
dev/	Folder	-	-	-
prod/	Folder	-	-	-

**Figure 1:** AWS S3 Backend - Keys

### Task 7: Change state backend configuration back to default

Update the terraform backend configuration block to it's default value by removing the `backend` block from the `terraform.tf` file, and migrate state back to a local state file.

`terraform.tf`

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
```





```
    source  = "hashicorp/aws"
    version = "~> 3.0"
}
http = {
    source  = "hashicorp/http"
    version = "2.1.0"
}
random = {
    source  = "hashicorp/random"
    version = "3.1.0"
}
local = {
    source  = "hashicorp/local"
    version = "2.1.0"
}
tls = {
    source  = "hashicorp/tls"
    version = "3.1.0"
}
}
```

```
terraform init -migrate-state
```

Initializing modules...

Initializing the backend...

Terraform has detected you're unconfiguring your previously set "s3" backend.

Do you want to copy existing state to the new backend?

Pre-existing state was found while migrating the previous "s3" backend to the

newly configured "local" backend. An existing non-empty state already exists in

the new backend. The two states have been saved to temporary files that will be removed after responding to this query.

Previous (type "s3"): /var/folders/1c/qvs1hwp964z\_dwd5qg07lv\_00000gn/T/terraform2246932369/1-s3.tfstate

New (type "local"): /var/folders/1c/qvs1hwp964z\_dwd5qg07lv\_00000gn/T/terraform2246932369/2-local.tfstate

Do you want to overwrite the state in the new backend with the previous state?

Enter "yes" to copy and "no" to start with the existing state in the newly





```
configured "local" backend.
```

```
Enter a value: yes
```

```
Successfully unset the backend "s3". Terraform will now operate locally.
```

```
terraform plan  
terraform apply
```

Once successfully migrated you can delete the partial state backend configurations, the `state_configuration` and `state_data` directories.





## Lab: Sensitive Data in Terraform State

Terraform state can contain sensitive data depending on the resources used. Sometimes it can contain initial database passwords or other secret data returned by a provider. Every time you deploy infrastructure with Terraform it stores lots of data about that infrastructure including all the parameters you passed in, within the state file.

We recommend that you protect Terraform state as you would any other sensitive piece of data.

- Task 1: View state file in raw format
- Task 2: Suppress sensitive information
- Task 3: View the Terraform state file

### Task 1: View Terraform State in Raw Format

By default the local state is stored in plain text as JSON. There is no additional encryption beyond your hard disk. View the Terraform state file of your most recent deployment:

```
terraform.tfstate
```

```
{  
  "version": 4,  
  "terraform_version": "1.1.1",  
  "serial": 28,  
  "lineage": "3f020a98-09d2-d827-e51d-673c7ec480bf",  
  "outputs": {  
    "public_dns": {  
      "value": "ec2-34-224-58-141.compute-1.amazonaws.com",  
      "type": "string"  
    },  
    "public_dns_server_subnet_1": {  
      "value": "ec2-3-239-20-18.compute-1.amazonaws.com",  
      "type": "string"  
    },  
    "public_ip": {  
      "value": "34.224.58.141",  
      "type": "string"  
    },  
    "public_ip_server_subnet_1": {  
      "value": "3.239.20.18",  
      "type": "string"  
    },  
    "size": {  
      "value": "t2.micro",  
      "type": "string"  
    }  
  }  
}
```





```

        "type": "string"
    }
},
"resources": [
{
    "mode": "data",
    "type": "aws_ami",
    "name": "ubuntu",
    "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
    "instances": [
        {
            ...

```

Other backends do support features like state encryption. With Amazon S3 for example you can enable encryption on the bucket, IAM access and logging, and only connect to state via TLS. Terraform Cloud also encrypts data in transit and at rest with a unique encryption key for each state version.

## Task 2: Suppress sensitive information

Regardless of where state is stored it should be treated like any other sensitive piece of data. To showcase this, add the following terraform code configuration to a new file in your working directory called `contactinfo.tf`

```

variable "first_name" {
    type = string
    sensitive = true
    default = "Terraform"
}

variable "last_name" {
    type = string
    sensitive = true
    default = "Tom"
}

variable "phone_number" {
    type = string
    sensitive = true
    default = "867-5309"
}

locals {
    contact_info = {
        first_name = var.first_name
        last_name = var.last_name
        phone_number = var.phone_number
    }
}

```





```

    }

    my_number = nonsensitive(var.phone_number)
}

output "first_name" {
    value = local.contact_info.first_name
}

output "last_name" {
    value = local.contact_info.last_name
}

output "phone_number" {
    value = local.contact_info.phone_number
}

output "my_number" {
    value = local.my_number
}

```

Execute a `terraform apply` with the variables in the `terraform.auto.tfvars`.

```
terraform apply
```

You will notice that several of the output blocks error as they need to have the `sensitive = true` value set.

```

| Error: Output refers to sensitive values

| on variables.tf line 73:
| 73: output "phone_number" {

| To reduce the risk of accidentally exporting sensitive data that was
| intended to be only internal, Terraform requires that any root module
| output containing
| sensitive data be explicitly marked as sensitive, to confirm your intent
|
| If you do intend to export this data, annotate the output value as
| sensitive by adding the following argument:
|   sensitive = true

```

Update the outputs to set the `sensitive = true` attribute and rerun the apply.

```

output "first_name" {
    sensitive = true
    value      = local.contact_info.first_name
}

```





```

}

output "last_name" {
  sensitive = true
  value     = local.contact_info.last_name
}

output "phone_number" {
  sensitive = true
  value     = local.contact_info.phone_number
}

```

`terraform apply`

Outputs:

```

first_name = <sensitive>
last_name = <sensitive>
my_number = "867-5309"
phone_number = <sensitive>

```

### Task 3: View the Terraform State File

Even though items are marked as sensitive within the Terraform configuration, they are stored within the Terraform state file. It is therefore critical to limit the access to the Terraform state file.

View the most current terraform state file:

```
{
  "version": 4,
  "terraform_version": "1.1.1",
  "serial": 29,
  "lineage": "3f020a98-09d2-d827-e51d-673c7ec480bf",
  "outputs": {
    "first_name": {
      "value": "Terraform",
      "type": "string",
      "sensitive": true
    },
    "last_name": {
      "value": "Tom",
      "type": "string",
      "sensitive": true
    },
    "my_number": {
      "value": "867-5309",
      "type": "string",
      "sensitive": true
    }
  }
}
```





```
        "type": "string"
    },
    "phone_number": {
        "value": "867-5309",
        "type": "string",
        "sensitive": true
    },
    ...
}
```

Once complete delete the `contactinfo.tf` configuration from your working directory, and run a `terraform destroy`.

## Terraform State Best Practices

### Treat State as Sensitive Data

We recommend that you protect Terraform state as you would any other sensitive piece of data. Depending on the Terraform resources, state may contain sensitive data including database passwords. This behavior is resource-specific and users should assume plain-text by default.

### Encrypt State Backend

Store Terraform state in a backend that supports encryption. Instead of storing your state in a local `terraform.tfstate` file, Terraform natively supports a variety of backends, such as S3, GCS, and Azure Blob Storage. Many of these backends support encryption, so that instead of your state files being in plain text, they will always be encrypted, both in transit (e.g., via TLS) and on disk (e.g., via AES-256).

### Control Access to State File

Strictly control who can access your Terraform backend. Since Terraform state files may contain secrets, you'll want to carefully control who has access to the backend you're using to store your state files. For example, if you're using S3 as a backend, you'll want to configure an IAM policy that solely grants access to the S3 bucket for production to a small handful of trusted devs (or perhaps solely just the CI server you use to deploy to prod).





## Lab: Local Values

A local value assigns a name to an expression, so you can use it multiple times within a configuration without repeating it. The expressions in local values are not limited to literal constants; they can also reference other values in the configuration in order to transform or combine them, including variables, resource attributes, or other local values.

You can use local values to simplify your Terraform configuration and avoid repetition. Local values (locals) can also help you write a more readable configuration by using meaningful names rather than hard-coding values. If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future. The ability to easily change the value in a central place is the key advantage of local values.

- Task 1: Create local values in a configuration block
- Task 2: Interpolate local values
- Task 3: Using locals with variable expressions
- Task 4: Using locals with terraform expressions and operators

### Task 1: Create local values in a configuration block

Add local values to your `main.tf` module directory:

```
locals {  
    service_name = "Automation"  
    app_team     = "Cloud Team"  
    createdby    = "terraform"  
}
```

### Task 2: Interpolate local values into your existing code

Update the `aws_instance` block inside your `main.tf` to add new tags to the `web_server` instance using interpolation.

```
...  
  
tags = {  
    "Service"  = local.service_name
```





```

    "AppTeam"   = local.app_team
    "CreatedBy" = local.createdby
}

...

```

After making these changes, rerun `terraform plan`. You should see that there will be some tagging updates to your server instances. Execute a `terraform apply` to make these changes.

### Task 3: Using locals with variable expressions

Expressions in local values are not limited to literal constants; they can also reference other values in the module in order to transform or combine them, including variables, resource attributes, or other local values.

Add another local values block to your `main.tf` module configuration that references the local values set in the previous portion of the lab.

```

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Name      = var.server_name
    Owner     = local.team
    App       = local.application
    Service   = local.service_name
    AppTeam   = local.app_team
    CreatedBy = local.createdby
  }
}

```

Update the `aws_instance` tags block inside your `main.tf` to reference the `local.common_tags` value.

```

resource "aws_instance" "web_server" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type                = "t2.micro"
  subnet_id                    = aws_subnet.public_subnets["public_subnet_1"].id
  ...
  tags = local.common_tags
}

```

After making these changes, rerun `terraform plan`. You should see that there are no changes to apply, which is correct, since the values contain the same values we had previously hard-coded, but now we are grabbing those values through the use of locals variables.





Note: You may see that the output mentions that `local_file.private_key_pem` will be created on each run, including this one but you can ignore that. Scroll up and note that there are no additional changes to the configuration.

```
...
```

```
No changes. Infrastructure is up-to-date.
```

```
This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, no actions need to be performed.
```





## Lab: Variables

We don't want to hardcode all of our values in the main.tf file. We can create a variable file for easier use. In the `variables` `block` lab, we created a few new variables, learned how to manually set their values, and even how to set the defaults. In this lab, we'll learn the other ways that we can set the values for our variables that are used across our Terraform configuration.

- Task 1: Set the value of a variable using environment variables
- Task 2: Declare the desired values using a tfvars file
- Task 3: Override the variable on the CLI

### Task 1: Set the value of a variable using environment variables

Often, the default values won't work and you will want to set a different value for certain variables. In Terraform OSS, there are 3 ways that we can set the value of a variable. The first way is setting an environment variable before running `terraform plan` or `terraform apply` command.

To set a value using an environment variable, we will use the `TF_VAR_` prefix, which is followed by the name of the variable. For example, to set the value of a variable named "variables\_sub\_cidr", we would need to set an environment variable called `TF_VAR_variables_sub_cidr` to the desired value.

On the CLI, use the following command to set an environment variable to set the value of our subnet CIDR block:

```
$ export TF_VAR_variables_sub_cidr="10.0.203.0/24"
```

#### Task 1.1

Run a `terraform plan` to see the results. You'll find that Terraform wants to replace the subnet since we updated the CIDR block of the subnet using an environment variable.

Note the value set in an environment variable takes precedence over the default value set in the variable block.

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
-/+ destroy and then create replacement

Terraform will perform the following actions:





```

# aws_subnet.terraform-subnet must be replaced
-/+ resource "aws_subnet" "terraform-subnet" {
    ~ arn                      = "arn:aws:ec2:us-east-
        -1:603991114860:subnet/subnet-0b424eed2dc2822d0" -> (known after
        apply)
    ~ availability_zone_id      = "use1-az6" -> (known after apply
        )
    ~ cidr_block                = "10.0.202.0/24" -> "
        10.0.203.0/24" # forces replacement
    ~ id                        = "subnet-0b424eed2dc2822d0" -> (
        known after apply)
    + ipv6_cidr_block_association_id = (known after apply)
    - map_customer_owned_ip_on_launch = false -> null
    ~ owner_id                  = "603991114860" -> (known after
        apply)
    tags                       = {
        "Name"       = "sub-variables-us-east-1a"
        "Terraform" = "true"
    }
    # (5 unchanged attributes hidden)
}

Plan: 1 to add, 0 to change, 1 to destroy.

```

## Task 1.2

Let's go ahead and apply our new configuration, which will replace the subnet with one using the CIDR block of “10.0.203.0/24”. Run a `terraform apply`. Don't forget to accept the changes by typing `yes`.

## Task 2: Declare the desired values using a tfvars file

Another way we can set the value of a variable is within a tfvars file. This is a special file that Terraform can use to retrieve specific values of variables without requiring the operator (you!) to modify the variables file or set environment variables. This is one of the most popular ways that Terraform users will set values in Terraform.

In the same Terraform directory, create a new file called `terraform.tfvars`. In that file, let's add the following code:

```
# Public Subnet Values
variables_sub_auto_ip = true
```





```
variables_sub_az      = "us-east-1d"
variables_sub_cidr   = "10.0.204.0/24"
```

## Task 2.1

Run a `terraform plan` to see the results. You'll find that Terraform wants to replace the subnet since we updated the CIDR block of the subnet using a `tfvars` file.

Note the value set in a `.tfvars` file takes precedence over an environment variable and the default value set in the variable block.

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
`-/+` destroy and then create replacement

Terraform will perform the following actions:

```
# aws_subnet.terraform-subnet must be replaced
-/+ resource "aws_subnet" "terraform-subnet" {
    ~ arn                      = "arn:aws:ec2:us-east-1:603991114860:subnet/subnet-0d9ef3f20d902ff28" -> (known after apply)
    ~ availability_zone          = "us-east-1a" -> "us-east-1d" #
    forces replacement
    ~ availability_zone_id       = "use1-az6" -> (known after apply)
    ~ cidr_block                = "10.0.203.0/24" -> "10.0.204.0/24" # forces replacement
    ~ id                         = "subnet-0d9ef3f20d902ff28" -> (known after apply)
    ~ known_after_apply
    + ipv6_cidr_block_association_id = (known after apply)
    - map_customer_owned_ip_on_launch = false -> null
    ~ owner_id                  = "603991114860" -> (known after apply)
    ~ tags                       = {
        ~ "Name"           = "sub-variables-us-east-1a" -> "sub-variables-us-east-1d"
        # (1 unchanged element hidden)
    }
    ~ tags_all                  = {
        ~ "Name"           = "sub-variables-us-east-1a" -> "sub-variables-us-east-1d"
        # (1 unchanged element hidden)
    }
    # (3 unchanged attributes hidden)
}
```





```
Plan: 1 to add, 0 to change, 1 to destroy.
```

## Task 2.2

Let's go ahead and apply our new configuration, which will replace the subnet with one using the CIDR block of "10.0.204.0/24". Run a `terraform apply`. Don't forget to accept the changes by typing `yes`.

## Task 3: Override the variable on the CLI

Finally, the last way that you can set the value for a Terraform variable is to simply set the value on the command line when running a `terraform plan` or `terraform apply` using a flag. You can set the value of a single variable using the `-var` flag, or you can set one or many variables using the `-var-file` flag and point to a file containing the variables and corresponding values.

On the CLI, run the following command:

```
$ terraform plan -var variables_sub_az="us-east-1e" -var  
variables_sub_cidr="10.0.205.0/24"
```

You'll see that we've now set the variable `variables_sub_az` equal to "us-east-1e" and the variable `variables_sub_cidr` to "10.0.205.0/24" which are different from our current infrastructure. As a result, Terraform wants to replace the existing subnet. Terraform uses the last value it finds, overriding any previous values.

Any values set on the CLI will take precedence over ANY other value set in a different way (ENV, tfvars, default value)

```
Terraform used the selected providers to generate the following execution  
plan. Resource actions are indicated with the following symbols:  
-/+ destroy and then create replacement
```

```
Terraform will perform the following actions:
```

```
# aws_subnet.terraform-subnet must be replaced  
-/+ resource "aws_subnet" "terraform-subnet" {  
    ~ arn                      = "arn:aws:ec2:us-east-  
    -1:603991114860:subnet/subnet-036f7e67555980f77" -> (known after  
    apply)
```





```

~ availability_zone           = "us-east-1d" -> "us-east-1e" #
  forces replacement
~ availability_zone_id        = "use1-az4" -> (known after apply)
  )
~ cidr_block                 = "10.0.204.0/24" -> "
  10.0.205.0/24" # forces replacement
~ id                          = "subnet-036f7e67555980f77" -> (
  known after apply)
+ ipv6_cidr_block_association_id = (known after apply)
- map_customer_owned_ip_on_launch = false -> null
~ owner_id                    = "603991114860" -> (known after
  apply)
~ tags                        = {
  ~ "Name"          = "sub-variables-us-east-1d" -> "sub-variables-us-
    east-1e"
    # (1 unchanged element hidden)
}
~ tags_all                     = {
  ~ "Name"          = "sub-variables-us-east-1d" -> "sub-variables-us-
    east-1e"
    # (1 unchanged element hidden)
}
# (3 unchanged attributes hidden)
}

```

Plan: 1 to add, 0 to change, 1 to destroy.

### Task 3.1

Let's go ahead and apply our new configuration, which will replace the subnet with one using the CIDR block of "10.0.204.0/24". Run a `terraform apply`. Don't forget to accept the changes by typing `yes`.





## Lab: Outputs

Terraform generates a significant amount of metadata that is too tedious to sort through with `terraform show`. Even with just one instance deployed, we wouldn't want to scan 38 lines of metadata every time. Outputs allow us to query for specific values rather than parse metadata in `terraform show`.

- Task 1: Create output values in the configuration file
- Task 2: Use the output command to find specific values
- Task 3: Suppress outputs of sensitive values in the CLI

### Task 1: Create output values in the configuration file

Outputs allow customization of Terraform's output during a Terraform apply. Outputs define useful values that will be highlighted to the user when Terraform is executed. Examples of values commonly retrieved using outputs include IP addresses, usernames, and generated keys.

Create a new output value named "public\_ip" to output the instance's public\_ip attributes. In the `outputs.tf` file, add the following:

```
output "public_ip" {  
  description = "This is the public IP of my web server"  
  value = aws_instance.web_server.public_ip  
}
```

#### Task 1.1: Run a Terraform Apply to view the outputs

After adding the new output blocks above, go ahead and run a `terraform apply -auto-approve` to see the new output values. Since we didn't change any resources, there is no risk of changes to our environment. You can see the new output value. Notice that you only see the name of the output and the value but you don't see the description in the output.

You can apply `this` plan to save these `new` output values to the Terraform state, without changing any real infrastructure.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

```
hello-world = "Hello World"
```





```
public_ip = "44.200.207.151"
public_url = "https://10.0.101.169:8080/index.html"
vpc_id = "vpc-0dcd2b053088ea107"
vpc_information = "Your demo_environment VPC has an ID of vpc-0
dcd2b053088ea107"
```

## Task 2: Use the `terraform output` command to find specific values

### Step 2.1 Try the `terraform output` command with no specifications

```
terraform output
```

### Step 2.2 Query specifically for the `public_dns` attributes

```
terraform output public_ip
```

### Step 2.3 Wrap an output query to ping the DNS record

```
ping $(terraform output -raw public_dns)
```

*Note that you will probably not receive a response since we don't have ICMP open on our security group*

## Task 3: Suppress outputs of sensitive values in the CLI

As you'll find with many aspects of Terraform, you will sometimes be working with sensitive data. Whether it's a username and password, account numbers, or certificates, it's very common that you'll want to obfuscate these from the CLI output. Fortunately, Terraform provides us with the `sensitive` argument to use in the output block. This allows you to mark the value as sensitive (hence the name) and prevent the value from showing in the CLI output. It does not, however, prevent the value from being listed in the state file or anything like that.

In the `outputs.tf` file, add the new output block as shown below. Since a resource arn often includes the AWS account number, it might be a value we don't want to show in the CLI console, so let's obfuscate it to protect our account.

```
output "ec2_instance_arn" {
  value = aws_instance.web_server.arn
```





```
sensitive = true  
}
```

### Task 3.1: Run a Terraform Apply to view the suppressed value

After adding the output blocks, run a `terraform apply -auto-approve` to see the new output value (or NOT see the new output value). Since we didn't change any resources, there is no risk of changes to our environment.

```
Changes to Outputs:  
  # (1 unchanged attribute hidden)  
  
You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
ec2_instance_arn = <sensitive>  
hello-world = "Hello World"  
public_ip = "44.200.207.151"  
public_url = "https://10.0.101.169:8080/index.html"  
vpc_id = "vpc-0dcd2b053088ea107"  
vpc_information = "Your demo_environment VPC has an ID of vpc-0dcd2b053088ea107"
```





## Lab: Variable Validation and Suppression

We may want to validate and possibly suppress and sensitive information defined within our variables.

- Task 1: Validate variables in a configuration block
- Task 2: More Validation Options
- Task 3: Suppress sensitive information
- Task 4: View the Terraform State File

### Task 1: Validate variables in a configuration block

Create a new folder called `variable_validation` with a `variables.tf` configuration file:

```
variable "cloud" {
  type = string

  validation {
    condition      = contains(["aws", "azure", "gcp", "vmware"], lower(var.cloud))
    error_message = "You must use an approved cloud."
  }

  validation {
    condition      = lower(var.cloud) == var.cloud
    error_message = "The cloud name must not have capital letters."
  }
}
```

Perform a `terraform init` and `terraform plan`. Provide inputs that both meet and do not meet the validation conditions to see the behavior.

```
terraform plan -var cloud=aws
terraform plan -var cloud=alibabba
```

### Task 2: More Validation Options

Add the following items to the `variables.tf`

```
variable "no_caps" {
  type = string
```





```

validation {
    condition = lower(var.no_caps) == var.no_caps
    error_message = "Value must be in all lower case."
}

variable "character_limit" {
    type = string

    validation {
        condition = length(var.character_limit) == 3
        error_message = "This variable must contain only 3 characters."
    }
}

variable "ip_address" {
    type = string

    validation {
        condition = can(regex("^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$", var.ip_address))
        error_message = "Must be an IP address of the form X.X.X.X."
    }
}

```

```

terraform plan -var cloud=aws -var no_caps=training
              -var ip_address=1.1.1.1 -var character_limit=rpt

terraform plan -var cloud=all -var no_caps=Training
              -var ip_address=1223.22.342.22 -var character_limit=ga

```

### Task 3: Suppress sensitive information

Terraform allows us to mark variables as sensitive and suppress that information. Add the following configuration into your `main.tf`:

```

variable "phone_number" {
    type = string
    sensitive = true
    default = "867-5309"
}

locals {
    contact_info = {
        cloud = var.cloud
    }
}

```





```

    department = var.no_caps
    cost_code = var.character_limit
    phone_number = var.phone_number
}

my_number = nonsensitive(var.phone_number)
}

output "cloud" {
  value = local.contact_info.cloud
}

output "department" {
  value = local.contact_info.department
}

output "cost_code" {
  value = local.contact_info.cost_code
}

output "phone_number" {
  value = local.contact_info.phone_number
}

output "my_number" {
  value = local.my_number
}

```

Execute a `terraform apply` with inline variables.

```
terraform apply -var cloud=aws -var no_caps=training
               -var ip_address=1.1.1.1 -var character_limit=rpt
```

You will notice that the output block errors as it needs to have the `sensitive = true` value set.

`Error: Output refers to sensitive values`

```
on variables.tf line 73:
73: output "phone_number" {
```

To reduce the risk of accidentally exporting sensitive data that was intended to be only internal, Terraform requires that any root module output containing sensitive data be explicitly marked as sensitive to confirm your intent.

If you **do** intend to **export** this data, annotate the output value as sensitive by adding the following argument:  
`sensitive = true`





Update the output to set the `sensitive = true` attribute and rerun the apply.

```
output "phone_number" {
  sensitive = true
  value = local.contact_info.phone_number
}
```

```
terraform apply -var cloud=aws -var no_caps=training
                 -var ip_address=1.1.1.1 -var character_limit=rpt
```

`Outputs:`

```
cloud = "aws"
cost_code = "rpt"
department = "training"
my_number = "867-5309"
phone_number = <sensitive>
```

#### Task 4: View the Terraform State File

Even though items are marked as sensitive within the Terraform configuration, they are stored within the Terraform state file. It is therefore critical to limit the access to the Terraform state file.

View the `terraform.tfstate` within your `variable_validation` directory.

```
{
  "version": 4,
  "terraform_version": "1.0.4",
  "serial": 3,
  "lineage": "5cfbccdd-b915-ee22-ea3c-17db83258332",
  "outputs": {
    "cloud": {
      "value": "aws",
      "type": "string"
    },
    "cost_code": {
      "value": "rpt",
      "type": "string"
    },
    "department": {
      "value": "training",
      "type": "string"
    },
    "my_number": {
      "value": "867-5309",
      "type": "string"
    }
  }
}
```





```
},
"phone_number": {
    "value": "867-5309",
    "type": "string",
    "sensitive": true
},
"resources": []
}
```

## TFC Integration

If you would like to see how variables are handled within Terraform Cloud, you can add the following files to your `variable_validation` directory.

`remote.tf`

```
terraform {
  backend "remote" {
    organization = "<>ORGANIZATION NAME><>"
    workspaces {
      name = "variable_validation"
    }
  }
}
```

`terraform.auto.tfvars`

```
cloud          = "aws"
no_caps        = "training"
ip_address     = "1.1.1.1"
character_limit = "rpt"
```

Run a `terraform init` to migrate state to the TFC workspace, followed by a `terraform apply` to show sensitive values with TFC.





## Lab: Secure Secrets in Terraform Code

When working with Terraform, it's very likely you'll be working with sensitive values. This lab goes over the most common techniques you can use to safely and securely manage such secrets.

- Task 1: Do Not Store Secrets in Plain Text
- Task 2: Mark Variables as Sensitive
- Task 3: Environment Variables
- Task 4: Secret Stores (e.g., Vault, AWS Secrets manager)

### Task 1: Do Not Store Secrets in Plain Text

Never put secret values, like passwords or access tokens, in .tf files or other files that are checked into source control. If you store secrets in plain text, you are giving the bad actors countless ways to access sensitive data. Ramifications for placing secrets in plain text include:

- Anyone who has access to the version control system has access to that secret.
- Every computer that has access to the version control system keeps a copy of that secret
- Every piece of software you run has access to that secret.
- No way to audit or revoke access to that secret.

### Task 2: Mark Variables as Sensitive

The first line of defense here is to mark the variable as sensitive so Terraform won't output the value in the Terraform CLI. Remember that this value will still show up in the Terraform state file:

In your `variables.tf` file, add the following code:

```
variable "phone_number" {  
    type      = string  
    sensitive = true  
    default   = "867-5309"  
}  
  
output "phone_number" {  
    value      = var.phone_number  
    sensitive = true  
}
```





Run a `terraform apply` to see the results of the sensitive variable. Notice how Terraform marks this as sensitive.

## Task 3: Environment Variables

Another way to protect secrets is to simply keep plain text secrets out of your code by taking advantage of Terraform's native support for reading environment variables. By setting the `TF_VAR_<name>` environment variable, Terraform will use that value rather than having to add that directly to your code.

In your `variables.tf` file, modify the `phone_number` variable and remove the default value so the sensitive value is no longer in cleartext:

```
variable "phone_number" { type = string sensitive = true }
```

In your terminal, export the following environment variable and set the value:

```
export TF_VAR_phone_number="867-5309"
```

*Note: If you are still using Terraform Cloud as your remote backend, you will need to set this environment variable in your Terraform Cloud workspace instead.*

Now, run a `terraform apply` and see that the plan runs just the same, since Terraform picked up the value of the sensitive variable using the environment variable. This strategy prevents us from having to add the value directly in our Terraform files and likely being committed to a code repository.

## Task 4: Inject Secrets into Terraform using HashiCorp Vault

Another way to protect your secrets is to store them in secrets management solution, like HashiCorp Vault. By storing them in Vault, you can use the Terraform Vault provider to quickly retrieve values from Vault and use them in your Terraform code.

Download HashiCorp Vault for your operating system at [vaultproject.io](https://vaultproject.io). Make sure the binary is moved to your `$PATH` so it can be executed from any directory. For help, check out <https://www.vaultproject.io/docs/install>. Alternatively, you can use Homebrew (MacOS) or Chocolatey (Windows). There are also RPMs available for Linux.

Validate you have Vault installed by running:

```
vault version
```





## Hands-On Labs

You should get back the version of Vault you have downloaded and installed.

In your terminal, run `vault server -dev` to start a Vault dev server. This will launch Vault in a pre-configured state so we can easily use it for this lab. Note that you should never run Vault in a production deployment by starting it this way.

Open a second terminal, and set the `VAULT_ADDR` environment variable. By default, this is set to HTTPS, but since we're using a dev server, TLS is not supported.

```
export VAULT_ADDR="http://127.0.0.1:8200"
```

Now, log in to Vault using the root token from the output of our Vault dev server. An example is below, but your root token and unseal key will be different:

**WARNING!** dev mode is enabled! In **this** mode, Vault runs entirely in-memory and starts unsealed with a single unseal key. The root token is already authenticated to the CLI, so you can immediately begin using Vault.

You may need to set the following environment variable:

```
$ export VAULT_ADDR='http://127.0.0.1:8200'
```

The unseal key and root token are displayed below in **case** you want to seal/unseal the Vault or re-authenticate.

Unseal Key: 1zqTTWCHyAvEhTq0LurR2nQPeeoR1Sk2FMp95fRNEaU=  
Root Token: s.Oi1tQPY98uwWQ6H0f9T7Elkg

**Development mode should NOT be used in production installations!**

Log in to Vault using the following command:

```
vault login <root token>
```

Now that we are logged into Vault, we can quickly add our sensitive values to be stored in Vault's KV store. Use the following command to write the sensitive value to Vault:

```
vault kv put /secret/app phone_number=867-5309
```

Back in Terraform, let's add the code to use Vault to retrieve our secrets. Create a new directory called `vault` and add a `main.tf` file. In your `main.tf` file, add the following code:

```
provider "vault" {  
  address = "http://127.0.0.1:8200"  
  token = <root token>  
}
```





By the way, note that I am only using the root token as an example here. Root tokens should NEVER be used on a day-to-day basis, nor should you use a root token for Terraform access. Please use a different auth method, such as AppRole, which is a better solution to establish connectivity between Terraform and Vault.

Now, add the following data block, which will use the Vault provider and token to retrieve the sensitive values we need:

```
data "vault_generic_secret" "phone_number" {
  path = "secret/app"
}
```

Finally, let's add a new output block that uses the data retrieved from Vault. In your main.tf, add the following code:

```
output "phone_number" {
  value = data.vault_generic_secret.phone_number
}
```

Run a `terraform init` and a `terraform apply`. Notice that Terraform is smart enough to understand that since the value was retrieved from Vault, it needs to be marked as sensitive since it likely contains sensitive information.

Add the `sensitive` configuration to the output block as follows:

```
output "phone_number" {
  value = data.vault_generic_secret.phone_number
  sensitive = true
}
```

Run a `terraform apply` again so Terraform retrieves that data from Vault. Note that the output will be shown as sensitive, but we can still easily display the data. In the terminal, run the following command to display the sensitive data retrieved from Vault:

```
terraform output phone_number
```

If you need ONLY the value of the data retrieved, rather than both the key and value and related JSON, you can update the output to the following:

```
output "phone_number" {
  value = data.vault_generic_secret.phone_number.data["phone_number"]
  sensitive = true
}
```

Run a `terraform output phone_number` again to see that the value is now ONLY the





phone\_number string, rather than the JSON output as we saw before.





## Lab: Terraform Collections and Structure Types

Duration: 10 minutes

As you continue to work with Terraform, you're going to need a way to organize and structure data. This data could be input variables that you are giving to Terraform, or it could be the result of resource creation, like having Terraform create a fleet of web servers or other resources. Either way, you'll find that data needs to be organized yet accessible so it is referenceable throughout your configuration. The Terraform language uses the following types for values:

- **string:** a sequence of Unicode characters representing some text, like “hello”.
- **number:** a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.
- **bool:** a boolean value, either true or false. bool values can be used in conditional logic.
- **list (or tuple):** a sequence of values, like [“us-west-1a”, “us-west-1c”]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
- **map (or object):** a group of values identified by named labels, like {name = “Mabel”, age = 52}. Maps are used to store key/value pairs.

Strings, numbers, and bools are sometimes called primitive types. Lists/tuples and maps/objects are sometimes called complex types, structural types, or collection types. Up until this point, we've primarily worked with string, number, or bool, although there have been some instances where we've provided a collection by way of input variables. In this lab, we will learn how to use the different collections and structure types available to us.

- Task 1: Create a new list and reference its values using the index
- Task 2: Add a new map variable to replace static values in a resource
- Task 3: Iterate over a map to create multiple resources
- Task 4: Use a more complex map variable to group information to simplify readability

### Task 1: Create a new list and reference its values

In Terraform, a *list* is a sequence of like values that are identified by an index number starting with zero. Let's create one in our configuration to learn more about it. Create a new variable that includes a list of different availability zones in AWS. In your `variables.tf` file, add the following variable:

```
variable "us-east-1-azs" {
  type = list(string)
  default = [
```





```

    "us-east-1a",
    "us-east-1b",
    "us-east-1c",
    "us-east-1d",
    "us-east-1e"
]
}

```

In your `main.tf` file, add the following code that will reference the new list that we just created:

```

resource "aws_subnet" "list_subnet" {
  vpc_id      = aws_vpc.vpc.id
  cidr_block  = "10.0.200.0/24"
  availability_zone = var.us-east-1-azs
}

```

Go and run a `terraform plan`. You should receive an error, and that's because the new variable `us-east-1-azs` we just created is a list of strings, and the argument `availability_zones` is expecting a single string. Therefore, we need to use an identifier to select which element to use in the list of strings.

Let's fix it. Update the `list_subnet` configuration to specify a specific element referenced by its indexed value from the list we provided - remember that indexes start at 0.

```

resource "aws_subnet" "list_subnet" {
  vpc_id      = aws_vpc.vpc.id
  cidr_block  = "10.0.200.0/24"
  availability_zone = var.us-east-1-azs[0]
}

```

Run a `terraform plan` again. Check out the output and notice that the new subnet will be created in `us-east-1a`, because that is the first string in our list of strings. If we used `var.us-east-1-azs[1]` in the configuration, Terraform would have built the subnet in `us-east-1b` since that's the second string in our list.

Go ahead and run `terraform apply` to apply the new configuration and build the subnet.

## Task 2 - Add a new map variable to replace static values in a resource

Let's continue to improve our `list_subnet` so we're not using any static values. First, we'll work on getting rid of the static value used for the subnet CIDR block and use a map instead. Add the following code to `variables.tf`:

```
variable "ip" {
```





```
type = map(string)
default = {
    prod = "10.0.150.0/24"
    dev  = "10.0.250.0/24"
}
}
```

Now, let's reference the new variable we just created. Modify the `list_subnet` in `main.tf`:

```
resource "aws_subnet" "list_subnet" {
    vpc_id      = aws_vpc.vpc.id
    cidr_block  = var.ip["prod"]
    availability_zone = var.us-east-1-azs[0]
}
```

Run `terraform plan` to see the proposed changes. In this case, you should see that the subnet will be replaced. The new subnet will have an IP subnet of 10.0.150.0/24, because we are now referencing the value of the `prod` key in our map.

Go ahead and run `terraform apply -auto-approve` to apply the new changes.

Before we move on, let's fix one more thing here. We still have a static value that is “hardcoded” in our `list_subnet` that we should use a variable for instead. We already have a `var.environment` that dictates the environment we’re working in, so we can simply use that in our `list_subnet`.

Modify the `list_subnet` in `main.tf` and update the `cidr_block` argument to the following:

```
resource "aws_subnet" "list_subnet" {
    vpc_id      = aws_vpc.vpc.id
    cidr_block  = var.ip[var.environment]
    availability_zone = var.us-east-1-azs[0]
}
```

Run `terraform plan` to see the proposed changes. In this case, you should see that the subnet will again be replaced. The new subnet will have an IP subnet of 10.0.250.0/24, because we are now using the value of `var.environment` to select the key in our map for the variable `var.ip` and the default is `dev`.

Go ahead and run `terraform apply -auto-approve` to apply the new changes.

### Task 3: Iterate over a map to create multiple resources

While we’re in much better shape for our `list_subnet`, we can still improve it. Oftentimes, you’ll want to deploy multiple resources of the same type but each resource should be slightly different for





different use cases. In our example, if we wanted to deploy BOTH a `dev` and `prod` subnet, we would have to copy the resource block and create a second one so we could refer to the other subnet in our map. However, there's a fairly simple way that we can iterate over our map in a single resource block to create and manage both subnets.

To accomplish this, use a `for_each` to iterate over the map to create multiple subnets in the same AZ. Modify your `list_subnet` to the following:

```
resource "aws_subnet" "list_subnet" {
  for_each          = var.ip
  vpc_id            = aws_vpc.vpc.id
  cidr_block        = each.value
  availability_zone = var.us-east-1-azs[0]
}
```

Run a `terraform plan` to see the proposed changes. Notice that our original subnet will be destroyed and Terraform will create two two subnets, one for `prod` with its respective CIDR block and one for `dev` with its respective CIDR block. That's because the `for_each` iterated over the map and will create a subnet for each key. The other major difference is the resource ID for each subnet. Notice how it's creating `aws_subnet.list_subnet["dev"]` and `aws_subnet.list_subnet["prod"]`, where the names are the keys listed in the map. This gives us a way to clearly understand what each subnet is. We could even use these values in a tag to name the subnet as well.

Go ahead and apply the new configuration using a `terraform apply -auto-approve`.

Using `terraform state list`, check out the new resources:

```
$ terraform state list
...
aws_subnet.list_subnet["dev"]
aws_subnet.list_subnet["prod"]
```

You can also use `terraform console` to view the resources and more detailed information about each one (use CTR-C to get out when you're done):

```
$ terraform console
> aws_subnet.list_subnet
{
  "dev" = {
    "arn" = "arn:aws:ec2:us-east-1:1234567890:subnet/subnet-052
           d26040d4b91a51"
    "assign_ipv6_address_on_creation" = false
  ...
}
```





#### Task 4: Use a more complex map variable to group information to simplify readability

While the previous configuration works great, we're still limited to using only a single availability zone for both of our subnets. What if we wanted to use a single resource block but have unique settings for each subnet? Well, we can use a map of maps to group information together to make it easier to iterate over and, more importantly, make it easier to read for you and others using the code.

Create a “map of maps” to group information per environment. In `variables.tf`, add the following variable:

```
variable "env" {
  type = map(any)
  default = {
    prod = {
      ip = "10.0.150.0/24"
      az = "us-east-1a"
    }
    dev = {
      ip = "10.0.250.0/24"
      az = "us-east-1e"
    }
  }
}
```

In `main.tf`, modify the `list_subnet` to the following:

```
resource "aws_subnet" "list_subnet" {
  for_each          = var.env
  vpc_id            = aws_vpc.vpc.id
  cidr_block        = each.value.ip
  availability_zone = each.value.az
}
```

Run a `terraform plan` to view the proposed changes. Notice that only the `dev` subnet will be replaced since we’re now placing it in a different availability zone, yet the `prod` subnet remains unchanged.

Go ahead and apply the configuration using `terraform apply -auto-approve`. Feel free to log into the AWS console to check out the new resources.

Once you’re done, feel free to delete the variables and `list_subnet` that was created in this lab, although it’s not required.





## Lab: Working with Data Blocks

Cloud infrastructure, applications, and services emit data, which Terraform can query and act on using data sources. Terraform uses data sources to fetch information from cloud provider APIs, such as disk image IDs, or information about the rest of your infrastructure through the outputs of other Terraform configurations.

- Task 1: Query existing resources using a data block
- Task 2: Export attributes from a data lookup

### Task 1: Query existing resources using a data block

As you develop Terraform code, you want to make sure the code is developed with reusability in mind. This often means that your code needs to query data in order to get specific attributes or values to deploy resources where needed. For example, if you manually created an S3 bucket in AWS, you might want to query information about that bucket so you can use it throughout your configuration. In this case, you would require a data block in Terraform to grab information to be used. Note that in this case, we're going to query data that already exists in AWS, and not a resource that was created by Terraform itself.

In the AWS console, create a new S3 bucket to use for this lab. Just create a bucket with all of the defaults. Don't worry, empty S3 buckets do not incur any costs.

In your `main.tf` file, let's create a data block that retrieves information about our new S3 bucket:

```
data "aws_s3_bucket" "data_bucket" {
  bucket = "my-data-lookup-bucket-btk"
}
```

Now, let's use information from that data lookup to create a new IAM policy to permit access to our new S3 bucket. In your `main.tf` file, add the following code:

```
resource "aws_iam_policy" "policy" {
  name      = "data_bucket_policy"
  description = "Deny access to my bucket"
  policy = jsonencode({
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "s3:Get*",

```





```

        "s3>List"
    ],
    "Resource": "${data.aws_s3_bucket.data_bucket.arn}"
}
})
}
}

```

Run a `terraform plan` to see the proposed changes. Notice how the new policy will be created and the resource in the policy is the ARN of our new S3 bucket. Go ahead and apply the configuration using a `terraform apply -auto-approve`.

## Task 2: Export attributes from a data lookup

Now that we have a successful data lookup against our S3 bucket, let's take a look at the attributes that we can export. Browse to the Terraform AWS provider, click on S3 in the left navigation page, and click on `aws_s3_bucket` under Data Sources. ([https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/s3\\_bucket](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/s3_bucket))

Note all of the different attributes that are exported. These are attributes that we can use throughout our Terraform configuration. We've already used `arn`, but we could use other attributes as well.

In the `main.tf` file, add the following outputs so we can see some of the additional information:

```

output "data-bucket-arn" {
  value = data.aws_s3_bucket.data_bucket.arn
}

output "data-bucket-domain-name" {
  value = data.aws_s3_bucket.data_bucket.bucket_domain_name
}

output "data-bucket-region" {
  value = "The ${data.aws_s3_bucket.data_bucket.id} bucket is located
          in ${data.aws_s3_bucket.data_bucket.region}"
}

```

Run a `terraform apply -auto-approve` to see the new outputs. Remember that the data in these outputs originated from our data lookup that we started with in this lab.

Once you're done, feel free to delete the bucket, the data block, and the output blocks if you would like.





## Lab: Terraform Built-in Functions

Duration: 15 minutes

As you continue to work with data inside of Terraform, there might be times where you want to modify or manipulate data based on your needs. For example, you may have multiple lists or strings that you want to combine. In contrast, you might have a list where you want to extract data, such as returning the first two values.

The Terraform language has many built-in functions that can be used in expressions to transform and combine values. Functions are available for actions on numeric values, string values, collections, date and time, filesystem, and many others.

- Task 1: Use basic numerical functions to select data
- Task 2: Manipulate strings using Terraform functions
- Task 3: View the use of cidrsubnet function to create subnets

### Task 1: Use basic numerical functions to select data

Terraform includes many different functions that work directly with numbers. To learn how they work, let's add some code and check it out. In your `variables.tf` file, let's add some variables. Feel free to use any number as the default value.

```
variable "num_1" {
  type = number
  description = "Numbers for function labs"
  default = 88
}

variable "num_2" {
  type = number
  description = "Numbers for function labs"
  default = 73
}

variable "num_3" {
  type = number
  description = "Numbers for function labs"
  default = 52
}
```

In the `main.tf`, let's add a new local variable that uses a numerical function:





```

locals {
  maximum = max(var.num_1, var.num_2, var.num_3)
  minimum = min(var.num_1, var.num_2, var.num_3, 44, 20)
}

output "max_value" {
  value = local.maximum
}

output "min_value" {
  value = local.minimum
}

```

Go ahead and run a `terraform apply -auto-approve` so we can see the result of our numerical functions by way of outputs.

## Task 2: Manipulate strings using Terraform functions

Now that we know how to use functions with numbers, let's play around with strings. Many of the resources we deploy with Terraform and the related arguments require a string for input, such as a subnet ID, security group, or instance size.

Let's modify our VPC to make use of a string function. Update your VPC resource in the `main.tf` file to look something like this:

```

#Define the VPC
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name      = upper(var.vpc_name)
    Environment = upper(var.environment)
    Terraform   = upper("true")
  }

  enable_dns_hostnames = true
}

```

Go ahead and run a `terraform apply -auto-approve` so we can see the result of our string functions by way of the changes that are applied to our tags.





## Task 2.1

Now, let's assume that we have set standards for our tags across AWS, and one of the requirements is that all tags are lower case. Rather than bothering our users with variable validations, we can simply take care of it for them with a simple function.

In your `main.tf` file, update the locals block to the following:

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Name      = lower(local.server_name)
    Owner     = lower(local.team)
    App       = lower(local.application)
    Service   = lower(local.service_name)
    AppTeam   = lower(local.app_team)
    CreatedBy = lower(local.createdby)
  }
}
```

Before we test it out, let's set the value of a variable using a `.tfvars` file. Create a new file called `terraform.auto.tfvars` in the same working directory and add the following:

```
environment = "PROD_Environment"
```

Let's test it out. Run `terraform plan` and let's take a look at the proposed changes. Notice that our string manipulations are causing some of the resource tags to be updated.

Go and apply the changes using `terraform apply -auto-approve`.

## Task 2.2

When deploying workloads in Terraform, it's common practice to use functions or expressions to dynamically generate name and tag values based on input variables. This makes your modules reusable without worrying about providing values for more and more tags.

In your `main.tf` file, let's update our locals block again, but this time we'll use a `join` to dynamically generate the value for Name based upon data we're already providing or getting from data blocks.

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Name = join("-", [
      local.application,
    ])
  }
}
```





```

        data.aws_region.current.name,
local.createdby])
Owner      = lower(local.team)
App        = lower(local.application)
Service    = lower(local.service_name)
AppTeam   = lower(local.app_team)
CreatedBy  = lower(local.createdby)
}
}

```

Let's test it out. Run `terraform plan` and let's take a look at the proposed changes. Notice that our string function is dynamically creating a Name for our resource based on other data we've provided or obtained.

Go and apply the changes using `terraform apply -auto-approve`.

### Task 3: View the use of cidrsubnet function to create subnets

There are many different specialized functions that come in handy when deploying resources in a public or private cloud. One of these special functions can help us automatically generate subnets based on a CIDR block that we provided it. Since the very first time you ran `terraform apply` in this course, you've been using the `cidrsubnet` function to create the subnets.

In your `main.tf` file, view the resource blocks that are creating our initial subnets:

```

#Deploy the private subnets
resource "aws_subnet" "private_subnets" {
  for_each          = var.private_subnets
  vpc_id            = aws_vpc.vpc.id
  cidr_block        = cidrsubnet(var.vpc_cidr, 8, each.value)
  availability_zone = tolist(data.aws_availability_zones.available.names) [
    each.value]

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}

```

Note how this block (and the one to create the public subnets) is creating multiple subnets. The resulting subnets were created based on the initial CIDR block, which was our VPC CIDR block. The second value is the number of bits added to the original prefix, so in our case, it was /16, so resulting subnets will be /24 since we're adding 8 in our function. The last value needed is derived from the





value obtained from the `value` of each key in `private_subnets` since we're using a `for_each` in this resource block.

Free free to create other subnets using the `cidrsubnet` function and play around with the values to see how it could best fit your requirements and scenarios.





## Lab: Dynamic Blocks

A dynamic block acts much like a for expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value. You can dynamically construct repeatable nested blocks using a special dynamic block type, which is supported inside resource, data, provider, and provisioner blocks.

- Task 1: Create a Security Group Resource with Terraform
- Task 2: Look at the state without a dynamic block
- Task 3: Convert Security Group to use dynamic block
- Task 4: Look at the state with a dynamic block
- Task 5: Use a dynamic block with Terraform map
- Task 6: Look at the state with a dynamic block using Terraform map

### Task 1: Create a Security Group Resource with Terraform

Add an AWS security group resource to our `main.tf`

```
resource "aws_security_group" "main" {  
    name      = "core-sg"  
    vpc_id    = aws_vpc.vpc.id  
  
    ingress {  
        description = "Port 443"  
        from_port   = 443  
        to_port     = 443  
        protocol    = "tcp"  
        cidr_blocks = ["0.0.0.0/0"]  
    }  
  
    ingress {  
        description = "Port 80"  
        from_port   = 80  
        to_port     = 80  
        protocol    = "tcp"  
        cidr_blocks = ["0.0.0.0/0"]  
    }  
}
```





## Task 2: Look at the state without a dynamic block

Run a `terraform apply` followed by a `terraform state list` to view how the security groups are accounted for in Terraform's State.

```
terraform state list
```

```
aws_security_group.main
```

```
terraform state show aws_security_group.main
```

```
# aws_security_group.main:
resource "aws_security_group" "main" {
    arn          = "arn:aws:ec2:us-east-1:508140242758:security-
                  group/sg-00157499a6de61832"
    description   = "Managed by Terraform"
    egress        = []
    id           = "sg-00157499a6de61832"
    ingress       = [
        {
            cidr_blocks = [
                "0.0.0.0/0",
            ]
            description = "Port 443"
            from_port   = 443
            ipv6_cidr_blocks = []
            prefix_list_ids = []
            protocol    = "tcp"
            security_groups = []
            self        = false
            to_port     = 443
        },
        {
            cidr_blocks = [
                "0.0.0.0/0",
            ]
            description = "Port 80"
            from_port   = 80
            ipv6_cidr_blocks = []
            prefix_list_ids = []
            protocol    = "tcp"
            security_groups = []
            self        = false
            to_port     = 80
        },
    ]
    name         = "core-sg"
```





```

    owner_id          = "508140242758"
    revoke_rules_on_delete = false
    tags_all          = {}
    vpc_id            = "vpc-0e3a3d76e5feb63c9"
}

```

### Task 3: Convert Security Group to use dynamic block

Refactor the `aws_security_group` resource block created above to utilize a dynamic block to built out the repeatable `ingress` nested block that is a part of this resource. We will supply the content for these repeatable blocks via local values to make it easier to read and update moving forward.

```

locals {
  ingress_rules = [
    {
      port      = 443
      description = "Port 443"
    },
    {
      port      = 80
      description = "Port 80"
    }
  ]
}

resource "aws_security_group" "main" {
  name    = "core-sg"
  vpc_id = aws_vpc.vpc.id

  dynamic "ingress" {
    for_each = local.ingress_rules

    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}

```





### Task 4: Look at the state with a dynamic block

Run a `terraform apply` followed by a `terraform state list` to view how the servers are accounted for in Terraform's State.

```
terraform apply  
terraform state list
```

```
aws_security_group.main
```

```
terraform state show aws_security_group.main
```

```
# aws_security_group.main:  
resource "aws_security_group" "main" {  
    arn          = "arn:aws:ec2:us-east-1:508140242758:security-  
                  group/sg-00157499a6de61832"  
    description   = "Managed by Terraform"  
    egress        = []  
    id           = "sg-00157499a6de61832"  
    ingress       = [  
        {  
            cidr_blocks = [  
                "0.0.0.0/0",  
            ]  
            description = "Port 443"  
            from_port   = 443  
            ipv6_cidr_blocks = []  
            prefix_list_ids = []  
            protocol     = "tcp"  
            security_groups = []  
            self         = false  
            to_port      = 443  
        },  
        {  
            cidr_blocks = [  
                "0.0.0.0/0",  
            ]  
            description = "Port 80"  
            from_port   = 80  
            ipv6_cidr_blocks = []  
            prefix_list_ids = []  
            protocol     = "tcp"  
            security_groups = []  
            self         = false  
            to_port      = 80  
        },  
    ],  
}
```





```

    name          = "core-sg"
    owner_id     = "508140242758"
    revoke_rules_on_delete = false
    tags          = {}
    tags_all     = {}
    vpc_id       = "vpc-0e3a3d76e5feb63c9"
}

```

### Task 5: Use a dynamic block with Terraform map

Rather than using the local values, we can refactor our dynamic block to utilize a variable named `web_ingress` which is of type map. Let's first create the variable of type map, specifying some default values for our ingress rules inside our `variables.tf` file.

```

variable "web_ingress" {
  type = map(object(
    {
      description = string
      port        = number
      protocol    = string
      cidr_blocks = list(string)
    }
  ))
  default = {
    "80" = {
      description = "Port 80"
      port        = 80
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
    "443" = {
      description = "Port 443"
      port        = 443
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}

```

Then we will refactor our security group to use this variable rather than using local values.

```

resource "aws_security_group" "main" {
  name = "core-sg"

  vpc_id = aws_vpc.vpc.id
}

```





```
dynamic "ingress" {
  for_each = var.web_ingress
  content {
    description = ingress.value.description
    from_port   = ingress.value.port
    to_port     = ingress.value.port
    protocol    = ingress.value.protocol
    cidr_blocks = ingress.value.cidr_blocks
  }
}
```

### Task 6: Look at the state with a dynamic block using Terraform map

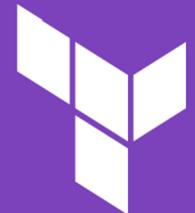
Run a `terraform apply` followed by a `terraform state list` to view how the servers are accounted for in Terraform's State.

```
terraform state list
```

```
terraform state show aws_security_group.main
```

```
# aws_security_group.main:
resource "aws_security_group" "main" {
  arn                  = "arn:aws:ec2:us-east-1:508140242758:security-
                        group/sg-00157499a6de61832"
  description          = "Managed by Terraform"
  egress               = []
  id                  = "sg-00157499a6de61832"
  ingress              = [
    {
      cidr_blocks       = [
        "0.0.0.0/0",
      ]
      description        = "Port 443"
      from_port          = 443
      ipv6_cidr_blocks  = []
      prefix_list_ids   = []
      protocol           = "tcp"
      security_groups    = []
      self                = false
      to_port             = 443
    },
    {
      cidr_blocks       = [
        "0.0.0.0/0",
      ]
    }
  ]
}
```





```
    description      = "Port 80"
    from_port       = 80
    ipv6_cidr_blocks = []
    prefix_list_ids = []
    protocol        = "tcp"
    security_groups = []
    self            = false
    to_port         = 80
  },
]
name          = "core-sg"
owner_id      = "508140242758"
revoke_rules_on_delete = false
tags          = {}
tags_all      = {}
vpc_id        = "vpc-0e3a3d76e5feb63c9"
}
```

## Best Practices

Overuse of dynamic blocks can make configuration hard to read and maintain, so it is recommend to use them only when you need to hide details in order to build a clean user interface for a re-usable module. Always write nested blocks out literally where possible.





## Lab: Terraform Graph

Terraform's interpolation syntax is very human-friendly, but under the hood it builds a very powerful resource graph. When resources are created they expose a number of relevant properties and Terraform's resource graph allows it to determine dependency management and order of execution for resource buildouts. Terraform has the ability to support the parallel management of resources because of its resource graph allowing it to optimize the speed of deployments.

The resource graph is an internal representation of all resources and their dependencies. A human-readable graph can be generated using the `terraform graph` command.

- Task 1: Terraform's Resource Graph and Dependencies
- Task 2: Generate a graph against Terraform configuration using `terraform graph`

### Task 1: Terraform's Resource Graph and Dependencies

When resources are created they expose a number of relevant properties. Let's look at portion of our `main.tf` that builds out our AWS VPC, private subnets, internet gateways and private keys. In this case, our private subnets and internet gateway are referencing our VPC ID and are therefore dependent on the VPC. Our private key however has no dependencies on any resources.

```
resource "aws_vpc" "vpc" {  
    cidr_block = var.vpc_cidr  
  
    tags = {  
        Name      = var.vpc_name  
        Environment = "demo_environment"  
        Terraform   = "true"  
    }  
  
    enable_dns_hostnames = true  
}  
  
resource "aws_subnet" "private_subnets" {  
    for_each           = var.private_subnets  
    vpc_id             = aws_vpc.vpc.id  
    cidr_block         = cidrsubnet(var.vpc_cidr, 8, each.value)  
    availability_zone = tolist(data.aws_availability_zones.available.names)[  
        each.value]  
  
    tags = {  
        Name      = each.key  
        Terraform = "true"  
    }  
}
```





```

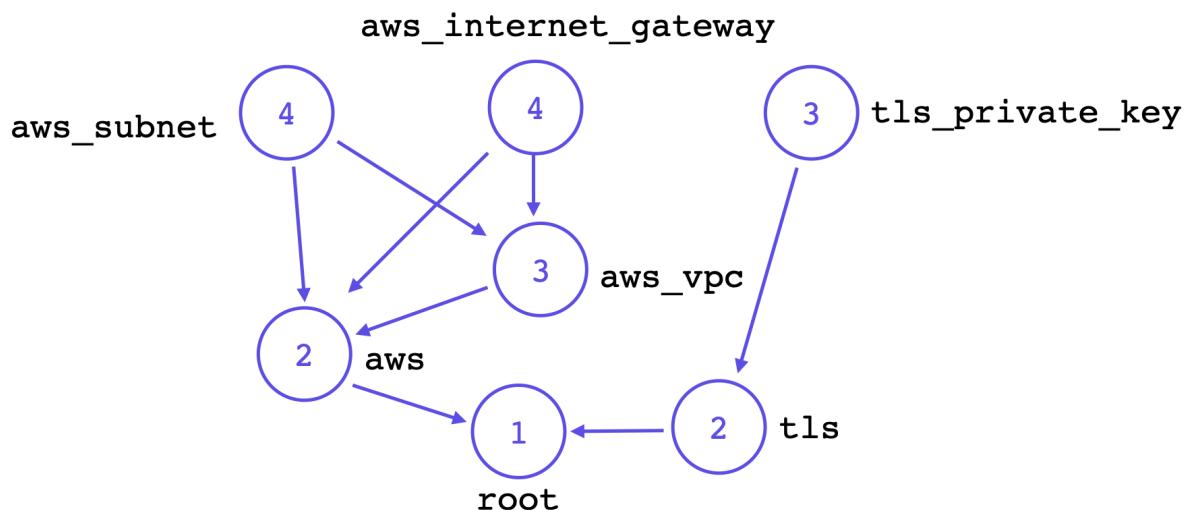
    }

resource "aws_internet_gateway" "internet_gateway" {
  vpc_id = aws_vpc.vpc.id
  tags = {
    Name = "demo_igw"
  }
}

resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

```

Because we have defined our infrastructure in code, we can build a data structure around it and then work with it. Some of these nodes depend on data from other nodes which need to be spun up first. Others might be disconnected or isolated. Our graph might look something like this, with the arrows showing the dependencies and order of operations.



**Figure 1:** Terraform Graph: Built-in dependency management

Terraform walks the graph several times starting at the root node and using the providers: to collect user input, to validate the config, to generate a plan, and to apply a plan. Terraform can determine which nodes need to be created sequentially and which can be created in parallel. In this case our private key can be built in parallel with our VPC, while our subnets and internet gateways are dependent on the AWS VPC being built first.





## Task 2: Generate a graph against Terraform configuration using `terraform graph`

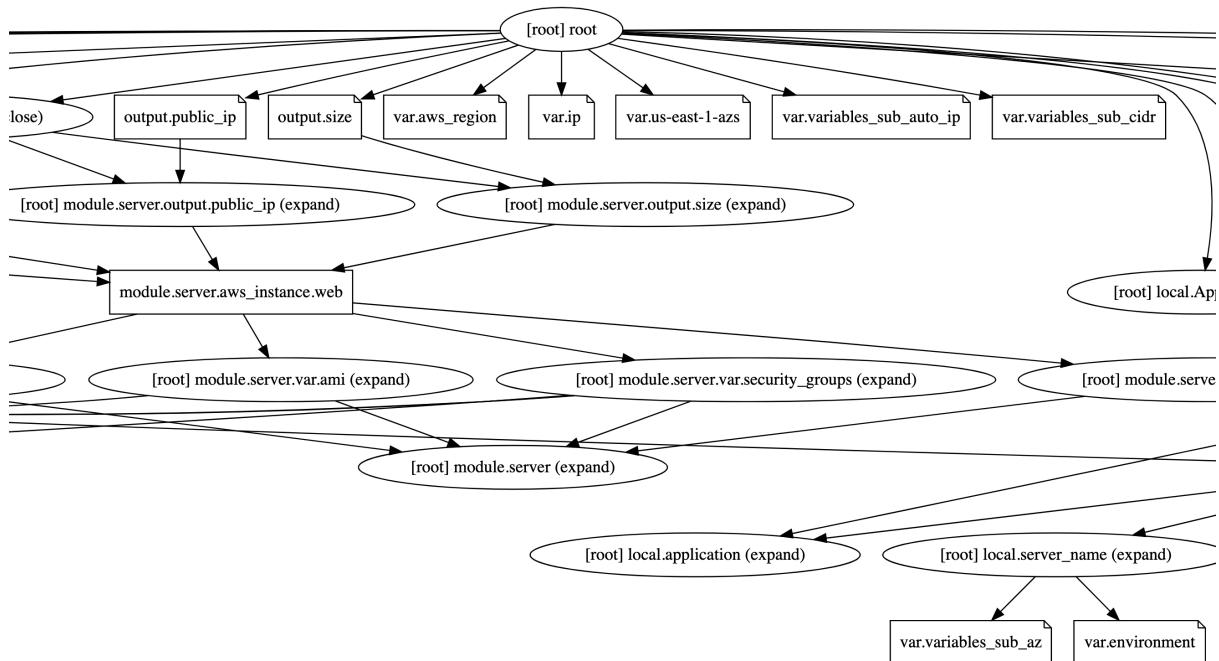
The `terraform graph` command is used to generate a visual representation of either a configuration or execution plan. The output is in the DOT format, which can be used by GraphViz to generate charts. This graph is useful for visualizing infrastructure and dependencies. Let's build out our infrastructure and use `terraform graph` to visualize the output.

```
terraform init
terraform apply
terraform graph
```

```
graph TD
    root([root] root)
    root --> output_public_ip[output.public_ip]
    root --> output_size[output.size]
    root --> var_aws_region[var.aws_region]
    root --> var_ip[var.ip]
    root --> var_us_east_1_azs[var.us-east-1-azs]
    root --> var_variables_sub_auto_ip[var.variables_sub_auto_ip]
    root --> var_variables_sub_cidr[var.variables_sub_cidr]

    # ...
}
```

Paste that output into <http://www.webgraphviz.com> to get a visual representation of dependencies that Terraform creates for your configuration.



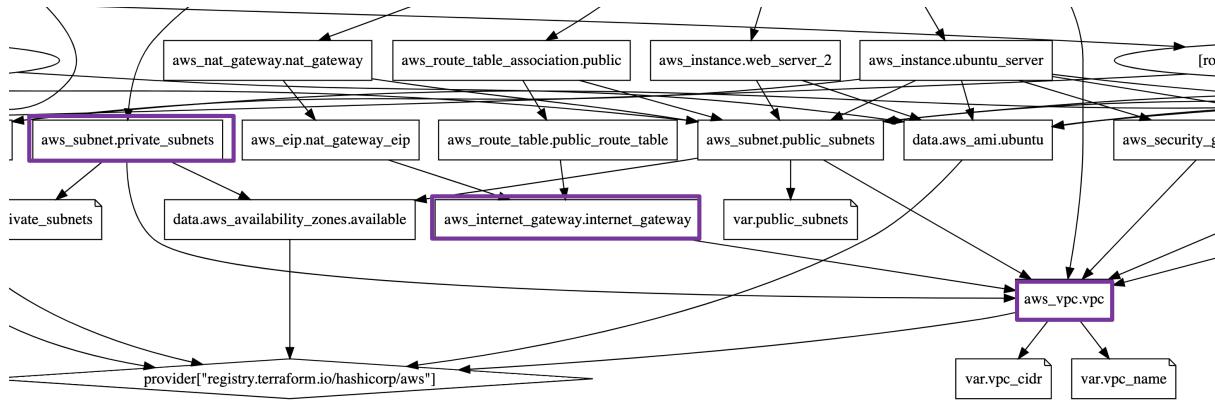
**Figure 2:** Visualize Terraform Graph





## Hands-On Labs

We can find our resources on the graph and follow the dependencies, which is what Terraform does everytime we exercise it's workflow.



**Figure 3:** Visualize Terraform Graph Dependencies





## Lab: Terraform Resource Lifecycles

Terraform has the ability to support the parallel management of resources because of its resource graph allowing it to optimize the speed of deployments. The resource graph dictates the order in which Terraform creates and destroys resources, and this order is typically appropriate. There are however situations where we wish to change the default lifecycle behavior that Terraform uses.

To provide you with control over dependency errors, Terraform has a [lifecycle](#) block. This lab demonstrates how to use lifecycle directives to control the order in which Terraform creates and destroys resources.

- Task 1: Use [create\\_before\\_destroy](#) with a simple AWS security group and instance
- Task 2: Use [prevent\\_destroy](#) with an instance

### Task 1: Use [create\\_before\\_destroy](#) with a simple AWS security group and instance

Terraform's default behavior when marking a resource to be replaced is to first destroy the resource and then create it. If the destruction succeeds cleanly, then and only then are replacement resources created. To alter this order of operation we can utilize the lifecycle directive [create\\_before\\_destroy](#) which does what it says on the tin. Instead of destroying an instance and then provisioning a new one with the specified attributes, it will provision first. So two instances will exist simultaneously, then the other will be destroyed.

Let's create a simple AWS configuration with a security group and an associated EC2 instance. Provision them with [terraform](#), then make a change to the security group. Observe that [apply](#) fails because the security group can not be destroyed and recreated while the instance lives.

You'll solve this situation by using [create\\_before\\_destroy](#) to create the new security group before destroying the original one.

#### 1.1: Add a new security group to the `security_groups` list of our server module

Add a new security group to the `security_groups` list of our server module by including `aws_security_group.main.id`

```
module "server_subnet_1" {
  source      = "./modules/web_server"
  ami         = data.aws_ami.ubuntu.id
  key_name    = aws_key_pair.generated.key_name
```





```

user          = "ubuntu"
private_key    = tls_private_key.generated.private_key_pem
subnet_id      = aws_subnet.public_subnets["public_subnet_1"].id
security_groups = [aws_security_group.vpc-ping.id, aws_security_group.
    ingress-ssh.id, aws_security_group.vpc-web.id, aws_security_group.
    main.id]
}

```

Initialize and apply the change to add the security group to our server module's `security_groups` list.

```

terraform init
terraform apply

```

### 1.2: Change the name of the security group

In order to see how some resources cannot be recreated under the default `lifecycle` settings, let's attempt to change the name of the security group from `core-sg` to something like `core-sg-global`.

```

resource "aws_security_group" "main" {
  name = "core-sg-global"

  vpc_id = aws_vpc.vpc.id

  dynamic "ingress" {
    for_each = var.web_ingress
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}

```

Apply this change.

```

terraform apply

```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:  
~ update **in-place**  
-/+ destroy and **then** create replacement





## Hands-On Labs

Terraform will perform the following actions:

```
# aws_security_group.main must be replaced
-/+ resource "aws_security_group" "main" {
    ~ arn                      = "arn:aws:ec2:us-east-1:508140242758:
                                security-group/sg-00157499a6de61832" -> (known after apply)
    ~ egress                   = [] -> (known after apply)
    ~ id                       = "sg-00157499a6de61832" -> (known after
                                apply)
    ~ name                     = "core-sg" -> "core-sg-global" # forces
                                replacement
    + name_prefix              = (known after apply)
    ~ owner_id                 = "508140242758" -> (known after apply)
    - tags                     = {} -> null
    ~ tags_all                 = {} -> (known after apply)
    # (4 unchanged attributes hidden)
}

# module.server_subnet_1.aws_instance.web will be updated in-place
~ resource "aws_instance" "web" {
    id                         = "i-0fbb3100e8671a855"
    tags                       = {
        "Environment" = "Training"
        "Name"       = "Web Server from Module"
    }
    ~ vpc_security_group_ids   = [
        - "sg-00157499a6de61832",
        - "sg-00dc379cbd0ad7332",
        - "sg-01fb306fc93cb941c",
        - "sg-0e0544dac3596af26",
    ] -> (known after apply)
    # (28 unchanged attributes hidden)

    # (5 unchanged blocks hidden)
}

Plan: 1 to add, 1 to change, 1 to destroy.
```

Notice that the default Terraform behavior is to destroy then create this resource is shown by the `-/+ destroy and then create replacement` statement.

Proceed with the apply.

Do you want to perform these actions?





Terraform will perform the actions described above.  
Only '`yes`' will be accepted to approve.

Enter a value: yes

**NOTE:** This action takes many minutes and eventually shows an error. You may choose to terminate the `apply` action with `^C` before the 15 minutes elapses. You may have to terminate twice to exit.

```
aws_security_group.main: Destroying... [id=sg-00157499a6de61832]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 10s
    elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 20s
    elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 30s
    elapsed]

...
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 14
    m40s elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 14
    m50s elapsed]
aws_security_group.main: Still destroying... [id=sg-00157499a6de61832, 15
    m0s elapsed]

| Error: Error deleting security group: DependencyViolation: resource sg
| -00157499a6de61832 has a dependent object
|     status code: 400, request id: 80dc904d-9439-41eb-8574-4b173685e72f
|
```

This is occurring because we have other resources that are dependent on this security group, and therefore the default Terraform behavior of destroying and then recreating the new security group is causing a dependency violation. We can solve this by using the `create_before_destroy` lifecycle directive to tell Terraform to first create the new security group before destroying the original.

### 1.3: Use `create_before_destroy`

Add a `lifecycle` configuration block to the `aws_security_group` resource. Specify that this resource should be created before the existing security group is destroyed.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  vpc_id = aws_vpc.vpc.id
```





```

dynamic "ingress" {
  for_each = var.web_ingress
  content {
    description = ingress.value.description
    from_port   = ingress.value.port
    to_port     = ingress.value.port
    protocol    = ingress.value.protocol
    cidr_blocks = ingress.value.cidr_blocks
  }
}

lifecycle {
  create_before_destroy = true
}

}

```

Now provision the new resources with the improved `lifecycle` configuration.

```
terraform apply
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

~ update **in-place**  
+/- create replacement and **then** destroy

Terraform will perform the following actions:

```

# aws_security_group.main must be replaced
+/- resource "aws_security_group" "main" {
  ~ arn                      = "arn:aws:ec2:us-east-1:508140242758:
                                security-group/sg-00157499a6de61832" -> (known after apply)
  ~ egress                    = [] -> (known after apply)
  ~ id                        = "sg-00157499a6de61832" -> (known after
                                apply)
  ~ name                      = "core-sg" -> "core-sg-global" # forces
                                replacement
  + name_prefix               = (known after apply)
  ~ owner_id                  = "508140242758" -> (known after apply)
  - tags                      = {} -> null
  ~ tags_all                  = {} -> (known after apply)
  # (4 unchanged attributes hidden)
}

# module.server_subnet_1.aws_instance.web will be updated in-place
~ resource "aws_instance" "web" {

```





## Hands-On Labs

```

    id                      = "i-0fbb3100e8671a855"
    tags                   = {
        "Environment" = "Training"
        "Name"        = "Web Server from Module"
    }
~ vpc_security_group_ids      = [
    - "sg-00157499a6de61832",
    - "sg-00dc379cbd0ad7332",
    - "sg-01fb306fc93cb941c",
    - "sg-0e0544dac3596af26",
] -> (known after apply)
# (28 unchanged attributes hidden)

# (5 unchanged blocks hidden)
}

```

**Plan:** 1 to add, 1 to change, 1 to destroy.

Notice now that the Terraform behavior is to create then destroy this resource is shown by the `+/- create replacement and then destroy` statement.

Proceed with the apply.

```

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

```

It should now succeed within a short amount of time as the new security group is created first, applied to our server and then the old security group is destroyed. Using the lifecycle block we controlled the order in which Terraform creates and destroys resources, removing the dependency violation of renaming a security group.

```

aws_security_group.main: Creating...
aws_security_group.main: Creation complete after 4s [id=sg-015
    f1eaa2c3f29f4c]
module.server_subnet_1.aws_instance.web: Modifying... [id=i-0
    fbb3100e8671a855]
module.server_subnet_1.aws_instance.web: Modifications complete after 5s [
    id=i-0fbb3100e8671a855]
aws_security_group.main (deposed object 5c96d2e2): Destroying... [id=sg
    -00157499a6de61832]

```





```
aws_security_group.main: Destruction complete after 1s
```

```
Apply complete! Resources: 1 added, 1 changed, 1 destroyed.
```

## Task 2: Use prevent\_destroy with an instance

Another lifecycle directive that we may wish to include in our configuration is `prevent_destroy`. This warns if any change would result in destroying a resource. All resources that this resource depends on must also be set to `prevent_destroy`. We'll demonstrate how `prevent_destroy` can be used to guard an instance from being destroyed.

### 2.1: Use prevent\_destroy

Add `prevent_destroy = true` to the same `lifecycle` stanza where you added `create_before_destroy`.

```
resource "aws_security_group" "main" {
  name = "core-sg-global"

  # ...

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
  }
}
```

Attempt to destroy the existing infrastructure. You should see the error that follows.

```
terraform destroy -auto-approve
```

```
Error: Instance cannot be destroyed
```

```
on main.tf line 378:
378: resource "aws_security_group" "main" {
```

Resource `aws_security_group.main` has `lifecycle.prevent_destroy` set, but the plan calls **for this** resource to be destroyed. To avoid **this** error and **continue** with the plan, either disable `lifecycle.prevent_destroy` or reduce the scope of the plan using the `-target` flag.





## 2.2: Destroy cleanly

Now that you have finished the steps in this lab, destroy the infrastructure you have created.

Remove the `prevent_destroy` attribute.

```
resource "aws_security_group" "main" {  
    name = "core-sg-global"  
  
    # ...  
  
    lifecycle {  
        create_before_destroy = true  
        # Comment out or delete this line  
        # prevent_destroy = true  
    }  
}
```

Finally, run `destroy`.

```
terraform destroy -auto-approve
```

The command should succeed and you should see a message confirming `Destroy complete!`

The `prevent_destroy` lifecycle directive can be used on resources that are stateful in nature (s3 buckets, RDS instances, long lived VMs, etc.) as a mechanism to help prevent accidentally destroying items that have long lived data within them.





## Lab: Terraform Cloud - Getting Started

Terraform Cloud is HashiCorp's managed service offering that eliminates the need for unnecessary tooling and documentation to use Terraform in production. Terraform Cloud helps you to provision infrastructure securely and reliably in the cloud with free remote state storage. Terraform Cloud and its self-hosted counterpart Terraform Enterprise offer Workspaces, Private Module Registry, Team Governance along with Policy as Code (Sentinel) as a few of its benefits.

- Task 1: Sign up for Terraform Cloud
- Task 2: Clone Getting Started Code Repository
- Task 3: Run setup script to create TFC Organization for course

### Task 1: Sign up for Terraform Cloud

In this lab you will sign up and get started with Terraform Cloud utilizing the Terraform Cloud.

1. Navigate to the sign up page and create an account for Terraform Cloud. If you already have a TFC account
2. Perform a `terraform login` from your workstation

Terraform will request an API token `for app.terraform.io` using your browser.

If `login` is successful, Terraform will store the token `in` plain text `in` the following file `for` use by subsequent commands:  
`/home/student/.terraform.d/credentials.tfrc.json`

Do you want to proceed?  
Only '`yes`' will be accepted to confirm.

Enter a value:

2. Answer `yes` at the prompt and generate a TFC user token by following the URL provided and copy-paste it into the prompt.

Open the following URL to access the tokens page `for app.terraform.io:`  
`https://app.terraform.io/app/settings/tokens?source=terraform-login`





1. If the token was entered successfully you should see the following:

```
Retrieved token for user tfcuser
```

```
Welcome to Terraform Cloud!
```

```
Documentation: terraform.io/docs/cloud
```

```
New to TFC? Follow these steps to instantly apply an example configuration:
```

```
$ git clone https://github.com/hashicorp/tfc-getting-started.git  
$ cd tfc-getting-started  
$ scripts/setup.sh
```





## Task 2: Clone Getting Started Code Repository

We will utilize a sample code repo to get started with Terraform Cloud. You can clone this sample repo using the following commands:

```
git clone https://github.com/hashicorp/tfc-getting-started.git  
cd tfc-getting-started
```

## Task 3: Run setup script to create TFC Organization for course

This startup script within the sample code repo automatically handles all the setup required to start using Terraform with Terraform Cloud. The included configuration provisions some mock infrastructure to a fictitious cloud provider called “Fake Web Services” using the `fakewebservices` provider.

```
./scripts/setup.sh
```

Follow the prompts provided by the script which will create a Terraform organization along with mock infrastructure.

```
-----  
Getting Started with Terraform Cloud  
-----
```

Terraform Cloud offers secure, easy-to-use remote state management and allows you to run Terraform remotely **in** a controlled environment. Terraform Cloud runs can be performed on demand or triggered automatically by various events.

This script will **set** up everything you need to get started. You'll be applying some example infrastructure - for free - in less than a minute.

First, we'll **do** some setup and configure Terraform to use Terraform Cloud.

Press any key to **continue** (ctrl-c to quit):

```
You did it! You just provisioned infrastructure with Terraform Cloud! The organization we created here has a 30-day free trial of the Team & Governance tier features.
```

Navigate to your workspace, along with mock infrastructure that you just deployed:

- Terraform Cloud: <https://app.terraform.io/>
- Mock infrastructure you just provisioned: <https://app.terraform.io/fake-web-services>





## Terraform Remote State - Enhanced Backend

Enhanced backends can both store state and perform operations. There are only two enhanced backends: `local` and `remote`. The `local` backend is the default backend used by Terraform which we worked with in previous labs. The `remote` backend stores Terraform state and may be used to run operations in Terraform Cloud. When using full remote operations, operations like `terraform plan` or `terraform apply` can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal. Remote plans and applies use variable values from the associated Terraform Cloud workspace.

- Task 1: Log in to Terraform Cloud
- Task 2: Update Terraform configuration to use Remote Enhanced Backend
- Task 3: Re-initialize Terraform and Validate Remote Backend
- Task 4: Provide Secure Credentials for Remote Runs
- Task 5: View the state, log and lock files in Terraform Cloud
- Task 6: Remove existing resources with `terraform destroy`

Let's take a closer look at the `remote` enhanced backend.

### Task 1: Log in to Terraform Cloud

Log in to Terraform Cloud and determine your organization name.

Note: You created a Terraform Cloud Organization in an earlier lab

Open the organization switcher menu in the top navigation bar and take note of your organization name.



A screenshot of the HashiCorp Terraform interface. On the left, there's a sidebar with a dark blue header containing the HashiCorp logo and the word "Organization". Below the header, the sidebar lists three organizations: "Enterprise-Cloud" (which is highlighted in a blue box), "Enterprise-DataCenter", and "example-org-5a4eda". To the right of the sidebar, the main area has a dark blue header with the text "Choose an organization ▾".

Choose an organization ▾

Enterprise-Cloud

Enterprise-DataCenter

example-org-5a4eda

**Figure 1:** Organization Name

## Task 2: Update Terraform configuration to use Remote Enhanced Backend

A configuration can only provide one backend block, so let's update our configuration to utilize the `remote` backend.

`terraform.tf`

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "YOUR-ORGANIZATION"  
  
    workspaces {  
      name = "my-aws-app"  
    }  
  }  
}
```

Update the `organization` name with what you noted in the previous step.





Example:

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "Enterprise-Cloud"  
  
    workspaces {  
      name = "my-aws-app"  
    }  
  }  
}
```

Note: A Terraform configuration can only specify a single backend. If a backend is already configured be sure to replace it. Copy just the backend block above and not the full terraform block. You can validate the syntax is correct by issuing a `terraform validate`

Be sure to issue a `terraform destroy` and delete any instances of the `terraform.tfstate` or `terraform.tfstate.backup` items locally as your state will be managed remotely.

### Task 3: Re-initialize Terraform and Validate Remote Backend

```
terraform init -reconfigure
```

Initializing the backend...

Successfully configured the backend "remote"! Terraform will automatically use `this` backend unless the backend configuration changes.

```
terraform apply
```

Running apply in the remote backend. Output will stream here. Pressing Ctrl-C will cancel the remote apply if it's still pending. If the apply started it will stop streaming the logs, but will not stop the apply running remotely.

Preparing the remote apply...

To view this run in a browser, visit:  
<https://app.terraform.io/app/Enterprise-Cloud/my-aws-app/runs/run-SThbeRjvZUUpH4e9>





Notice that the Terraform apply with the `remote` backend looks different than with the standard backends. The enhanced Terraform `remote` backend stores the state information within Terraform Cloud as well as performs all operations from Terraform Cloud. Therefore this backend supports the ability to centrally store state and centrally manage the Terraform workflow.

This can be validated by the messages returned when executing Terraform CLI commands using the `remote` backend.

#### Task 4: Provide Secure Credentials for Remote Runs

Now that the terraform workflow is being run using the `remote` backend inside Terraform Cloud, we have to configure Terraform Cloud to use our AWS Credentials for building out our infrastructure. In fact during our last step you most likely encountered an error similar to the following:

```
Error: error configuring Terraform AWS Provider: no valid credential sources for Terraform AWS Provider found.

Please see https://registry.terraform.io/providers/hashicorp/aws
for more information about providing credentials.
```

We can provide Terraform Cloud with our AWS Credentials as environment variables inside the `Variables` section of our workspace.

Enterprise-Cloud / Workspaces / my-aws-app / Variables

my-aws-app	Resources 0	Terraform version 1.0.10	Updated 3 hours ago
No workspace description available. <a href="#">Add workspace description</a> .			
<a href="#">Overview</a> <a href="#">Runs</a> <a href="#">States</a> <b>Variables</b> <a href="#">Settings</a> <span style="float: right;"><a href="#">Actions</a> <span style="font-size: small;">Unlocked</span> </span>			

**Variables**

Terraform uses all [Terraform](#) and [Environment](#) variables for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration. You may want to use the Terraform Cloud Provider or the variables API to add multiple variables at once.

**Sensitive variables**

[Sensitive](#) variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

**Workspace variables (0)**

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence [»](#).

Key	Value	Category
There are no variables added.		
<a href="#">+ Add variable</a>		

**Figure 2:** Workspace Variables



## Hands-On Labs



Select [Add variable](#), Select [Environment variable](#). Create both a `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variable with the AWS credentials. Now whenever the Terraform workflow is executed using our configured `remote` backend, Terraform Cloud knows which credentials to use to access our AWS cloud account.

You can mark one or both of these variables as [Sensitive](#) so that others cannot view their values.

[my-aws-app](#)

No workspace description available. [Add workspace description](#).

Overview	Runs	States	Variables	Settings	Resources	Terraform version	Updated
					0	1.0.10	3 hours ago
						Unlocked	<a href="#">Actions</a>

### Variables

Terraform uses all [Terraform](#) and [Environment](#) variables for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration. You may want to use the Terraform Cloud Provider or the variables API to add multiple variables at once.

#### Sensitive variables

[Sensitive](#) variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

#### Workspace variables (2)

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence [»](#).

Key	Value	Category	...
<code>AWS_ACCESS_KEY_ID</code>	AKIAJAXMT4EY5DLVHLRJH6	env	...
<code>AWS_SECRET_ACCESS_KEY</code> SENSITIVE	Sensitive - write only	env	...

[+ Add variable](#)

**Figure 3:** TFC AWS Variables

Once set you can build out the infrastructure

```
terraform apply

Running apply in the remote backend. Output will stream here. Pressing
Ctrl-C
will cancel the remote apply if it's still pending. If the apply started
it
will stop streaming the logs, but will not stop the apply running remotely
.

Preparing the remote apply...

To view this run in a browser, visit:
https://app.terraform.io/app/Enterprise-Cloud/my-aws-app/runs/run-2
uxkrzPJ62pmHrQ6

Waiting for the plan to start...
```



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



```
Terraform v1.0.10  
on linux_amd64  
Configuring remote state backend...  
Initializing Terraform configuration...  
....  
Plan: 27 to add, 0 to change, 0 to destroy.  
  
Do you want to perform these actions in workspace "my-aws-app"?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.  
  
Enter a value: yes
```

You can view the apply also within the Terraform Cloud UI.

The screenshot shows the Terraform Cloud workspace interface. At the top, there's a summary card for the workspace 'my-aws-app': Resources 0, Terraform version 1.0.10, and Updated a few seconds ago. Below this, a navigation bar includes Overview, **Runs**, States, Variables, and Settings. The 'Runs' tab is selected. A 'Current Run' section shows a single run triggered via CLI by user gabe\_maentz, labeled as CURRENT and Applying, triggered a few seconds ago. Below it, a 'Run List' section shows the same run again, also triggered via CLI by gabe\_maentz, labeled as CURRENT and Applying, triggered a few seconds ago.

**Figure 4:** Terraform Cloud Apply



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

my-aws-app

No workspace description available. [Add workspace description](#).

Resources 0      Terraform version 1.0.10      Updated a few seconds ago

Overview    **Runs**    States    Variables    Settings    Actions

**Triggered via CLI**

**CURRENT**

gabe\_maentz triggered a run from CLI 2 minutes ago

Plan finished 2 minutes ago      Resources: 27 to add, 0 to change, 0 to destroy

Apply running a minute ago      Started a minute ago

[View raw log](#)

```
aws_instance.ubuntu_server: Creating...
aws_subnet_public_subnets["public_subnet_2"]: Creation complete after 12s [id=subnet-05205168af0a8552f]
aws_route_table_association.public["public_subnet_2"]": Creating...
aws_nat_gateway.nat_gateway: Creating...
aws_route_table_association.public["public_subnet_1"]": Creating...
aws_route_table_association.public["public_subnet_3"]": Creating...
aws_instance.web_server_2: Creating...
aws_route_table_association.public["public_subnet_1"]": Creation complete after 0s [id=rtbassoc-0ef85fa24f466c773]
aws_route_table_association.public["public_subnet_3"]": Creation complete after 1s [id=rtbassoc-0d6d4ba8757430ace]
aws_instance.ubuntu_server: Still creating... [10s elapsed]
aws_route_table_association.public["public_subnet_1"]": Still creating... [10s elapsed]
aws_nat_gateway.nat_gateway: Still creating... [10s elapsed]
aws_instance.web_server_2: Still creating... [10s elapsed]
aws_instance.ubuntu_server: Still creating... [20s elapsed]
aws_route_table_association.public["public_subnet_1"]": Still creating... [20s elapsed]
aws_instance.web_server_2: Still creating... [20s elapsed]
aws_instance.ubuntu_server: Still creating... [30s elapsed]
aws_route_table_association.public["public_subnet_1"]": Still creating... [30s elapsed]
aws_instance.web_server_2: Still creating... [30s elapsed]
aws_instance.ubuntu_server: Still creating... [40s elapsed]
aws_route_table_association.public["public_subnet_1"]": Still creating... [40s elapsed]
aws_instance.web_server_2: Still creating... [40s elapsed]
aws_instance.ubuntu_server: Creation complete after 42s [id=i-0dba0a3c2a5be1bf6]
aws_instance.ubuntu_server: Still creating... [50s elapsed]
```

**Figure 5:** Terraform Cloud Apply Detail

### Task 5: View the state, log and lock files in Terraform Cloud

Using the `remote` backend allows you to connect to Terraform Cloud. Within the Terraform Cloud UI you'll be able to:

- View Terraform run history
- View state history
- View all your organization's workspaces
- Lock a workspace, making it easy to avoid conflicting changes and state corruption
- Execute the Terraform workflow via CLI or UI

Let's view the state file in Terraform Cloud.

- If you aren't already, log in to Terraform Cloud in your browser
- Navigate to your Workstation
- In your Workstation UI page, click on the [States](#) tab





## Hands-On Labs

- You should see something along the lines of this in the [States](#) tab. This indicates that your state file is now being stored and versioned within Terraform Cloud using the [remote](#) backend.

my-aws-app

No workspace description available. [Add workspace description](#).

Overview Runs States Variables Settings Actions

Resources 30 Terraform version 1.0.10 Updated a few seconds ago

Triggered via CLI #sv-2YVbchjpvDVND2w | gabe\_maentz triggered from Terraform | #run-2uxkrzPJ62pmHrQ6

Download 3 minutes ago

filter Apply Learn more about filtering JSON data. Expand Full screen

```
1 {
2   "version": 4,
3   "terraform_version": "1.0.10",
4   "serial": 0,
5   "lineage": "ae54c6de-88ca-568c-39af-68b6f4c8511f",
6   "outputs": {},
7   "resources": [
8     {
9       "mode": "data",
10      "type": "aws_ami",
11      "name": "ubuntu",
12      "provider": "provider{\"registry.terraform.io/hashicorp/aws\":}",
13      "instances": [
14        {
15          "schema_version": 0,
16          "attributes": {
17            "architecture": "x86_64",
18          }
19        }
20      ]
21    }
22  ]
23}
```

## Task 6: Remove existing resources with `terraform destroy`

We will now issue a cleanup of our infrastructure using a [terraform destroy](#)

```
terraform destroy
```

```
Plan: 0 to add, 0 to change, 27 to destroy.
```

```
Do you really want to destroy all resources?
```

```
Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only 'yes' will be accepted to confirm.
```

```
Enter a value: yes
```





## Lab: Terraform Cloud Workspaces

A Terraform workspace is a managed unit of infrastructure. Workspaces are the workhorse of Terraform Cloud and build on the Terraform CLI workspace construct. Each uses the same Terraform code to deploy infrastructure and each keeps separate state data for each workspace. Terraform Cloud simply adds more functionality. On your local workstation, the terraform workspace is simply a directory full of terraform code and variables. This code is also ideally stored in a git repository. Terraform Cloud workspaces take on some extra roles. In Terraform Cloud your workspace stores state data, has its own set variable values and environment variables, and allows for remote operations and logging. Terraform Cloud workspaces also provide access controls, version control integration, API access and policy management.

This lab demonstrates how to utilize workspaces within Terraform Cloud.

- Task 1: Review Terraform Cloud Workspaces User Interface
- Task 2: Create a Terraform Cloud Workspace for DEV deployments
- Task 3: Create a Terraform Cloud Workspace for PROD developments
- Task 4: Assign and set variables per workspace
- Task 5: Change Terraform versions per workspace
- Task 6: Deploy infrastructure using Terraform Cloud workspaces

### Task 1: Review Terraform Cloud Workspaces Features

Now that we have our state and variables stored in our Terraform Cloud workspace and have begun to execute remote runs within the workspace, let's look at some of the other features that Terraform Cloud workspaces have to offer.

#### 1.1 Workspace Dashboard

The landing page for a workspace includes an overview of its current state and configuration.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

The screenshot shows the 'Overview' tab of a Terraform Cloud workspace named 'my-aws-app'. At the top right, there's a summary card with 'Resources 30', 'Terraform version 1.0.11', and 'Updated a few seconds ago'. A purple arrow points from the top right towards this card. Below the card, there are execution mode and auto-apply settings: 'Execution mode: Remote' (unlocked) and 'Auto apply: Off'. Another purple arrow points from the left towards the 'Latest Run' section. The 'Latest Run' card shows it was triggered via CLI by 'gabe\_maentz' an hour ago, with 2 of 2 policy checks passed, an estimated cost increase of '\$25.89', a plan & apply duration of 'Less than a minute', and 0 resources changed. A purple arrow points from the bottom left towards the resource list. The resource list table has columns: NAME, PROVIDER, TYPE, MODULE, and UPDATED. It shows one row for 'available' with provider 'hashicorp/aws', type 'data.aws\_ava...', module 'root', and updated on 'Dec 28 2021'. A purple arrow points from the bottom right towards the metrics and tags sections. The metrics section displays average plan duration (< 1 min), average apply duration (2 mins), total failed runs (3), and policy check failures (1). The tags section shows '(0)' and an 'Add a tag' button.

**Figure 1:** Terraform Cloud Workspace - Overview

You quickly find resource count, terraform version, and the time since the last update. You can view the latest information about last run including who performed the run, how the run was triggered, which resources changed, the duration of the run, along with the cost change. For further details you can jump into the he Runs tab to see all historical runs.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



my-aws-app

No workspace description available. [Add workspace description.](#)

Resources: 30 | Terraform version: 1.0.11 | Updated: an hour ago

Overview Runs States Variables Settings Actions

Current Run

	Triggered via CLI	CURRENT		an hour ago
#run-jNpYaBjyaQuhhca8   gabe_maentz triggered via CLI				

Run List

	Triggered via CLI	CURRENT		an hour ago
#run-VfCHHjr7cENxztTB   gabe_maentz triggered via UI				
	Triggered via CLI	#run-tEdxuRFQRzSV4HSd   gabe_maentz triggered via CLI		3 hours ago

**Figure 2:** Terraform Cloud Workspace - Historical Runs

The **Actions** menu allows you to lock the workspace or trigger a new run. The resources and outputs tables display details about the current infrastructure and any outputs configured for the workspace. The sidebar contains metrics and details about workspace settings. It also includes a section for workspace tags. You can manage your workspace's tags here, and your list of tags in your Organization settings. Tags allow you to group and filter your workspaces.

### 1.2 Remote or Local Runs

As we have shown by using the `remote` backend, Terraform operations like `plan` or `apply` can now be run within Terraform Cloud on a hosted agent which saves the results and logs to the workspace. This gives you a standardized working environment for all Terraform runs and possibly more importantly, a single place to go and inspect logs when something goes wrong.





## General Settings

**ID**

ws-3PyarudNZ5dY1h2R

**Name**

my-aws-app

**Description**

Optional

Workspace description

**Execution Mode**

If you change the execution mode any in progress runs will be discarded.

 **Remote**

Your plans and applies occur on Terraform Cloud's infrastructure. You and your team have the ability to review and collaborate on runs within the app.

 **Local**

Your plans and applies occur on machines you control. Terraform Cloud is only used to store and synchronize state.

**Figure 3:** Terraform Cloud Workspace - Execution Mode

You can also choose between remote and local execution on a per workspace basis.

### 1.3 Terraform Version

When operating in Remote execution mode we can also specify the version of Terraform for the hosted agent to run without having to worry about the Terraform version installed on the local machine.



**Manual apply**

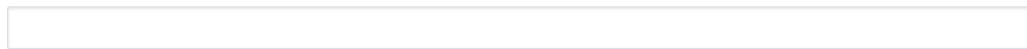
Require an operator to confirm the result of the Terraform plan before applying. If this workspace is linked to version control, a push to the default branch of the linked repository will only trigger a plan and then wait for confirmation.

**Terraform Version**

1.0.11



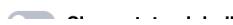
The version of Terraform to use for this workspace. Upon creating this workspace, the latest version was selected and will be used until it is changed manually. It will **not upgrade automatically**.

**Terraform Working Directory**

The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

**Remote state sharing**

Choose whether this workspace should share state with the entire organization, or only with specific approved workspaces. The `terraform_remote_state` data source relies on state sharing to access workspace outputs.

**Figure 4:** Terraform Cloud Workspace - Terraform Version

The Terraform core version can be specified on a per workspace basis and can differ between workspaces. Workspaces make it easy to standardize and upgrade when the time comes.

#### 1.4 Team Access

Each workspace can be configured with team access permissions. The Team Access category allows you to define which teams have access to the workspace and what level of access they have. There are pre-canned levels of access, like read, plan, write, and admin, or you can construct your own custom permission sets.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

my-aws-app

No workspace description available. [Add workspace description](#).

Overview Runs States Variables **Settings** Actions

Resources 30 Terraform version 1.0.11 Updated an hour ago

**Team Access** Add team and permissions

**Heads up**  
Teams with [organization-level permissions](#) can also access this workspace, even if they are not listed on this page or are listed at a lower access level.

NAME	PRIVILEGES
Owners of example-org-6cde13	default

**Figure 5:** Terraform Cloud Workspace - Team Access



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



### Add Team Permissions

Add a team and assign permissions to this workspace.

Select a team

2 Assign permissions

#### Assign permissions to `tfc_getting_started`

Assign permissions to the selected team below.

Customize permissions for this team

##### Read

[Assign permissions](#)

###### Baseline permissions for reading a workspace

- |                              |                  |                           |
|------------------------------|------------------|---------------------------|
| ✓ Read runs                  | ✓ Read variables | ✓ Read TF config versions |
| ✓ Read workspace information | ✓ Read state     |                           |

##### Plan

[Assign permissions](#)

###### Read permissions plus the ability to create runs

- |                           |               |
|---------------------------|---------------|
| ✓ All permissions of read | ✓ Create runs |
|---------------------------|---------------|

##### Write

[Assign permissions](#)

###### Read, plan and write permissions

- |                           |                      |                |
|---------------------------|----------------------|----------------|
| ✓ All permissions of plan | ✓ Can read and write | ✓ Approve runs |
| ✓ Lock/unlock workspace   |                      |                |

##### Admin

[Assign permissions](#)

###### Full control of the workspace

- |                            |                      |                    |
|----------------------------|----------------------|--------------------|
| ✓ All permissions of write | ✓ Manage team access | ✓ Delete workspace |
| ✓ VCS configuration        | ✓ Execution mode     | ✓ Access to state  |

**Figure 6:** Terraform Cloud Workspace - Team Access

### 1.5 Notifications

Moving over to the notifications category, when something happens with the workspace, you can trigger a notification to be sent to an email address, slack channel, or webhook. This is great if you want to be notified when a new plan needs to be approved or an apply went horribly wrong.





## Create a Notification

Notifications allow you to send messages to other applications based on Run events.

### Destination

 <b>Webhook</b> POST messages to any URL  <input type="radio"/>	 <b>Email</b> Send messages to users via Email  <input type="radio"/>	 <b>Slack</b> Send messages to a Slack Channel  <input type="radio"/>
--	--	--

### Name

### Webhook URL

### Token

Used to generate the HMAC on the notification request. [Read more in the documentation](#).

### Triggers

Choose the events you want to receive notifications on.

**All events**

**Only certain events**

**Created**

Every time a run is created and enters the "Pending" state.

**Planning**

When a run acquires the lock and starts to execute.

**Needs Attention**

Human decision required. When a plan has changes and is not auto-applied, or requires a policy override.

**Applying**

After a plan is confirmed or auto-applied.

**Completed**

When the run has completed on a happy path and can't go any further.

**Errored**

If the run has terminated early due to error or cancelation.

**Create a notification**

**Cancel**

**Figure 7:** Terraform Cloud Workspace - Notifications





## 1.6 Workflows: CLI/VCS/API Integration

When you create a new workspace, Terraform Cloud will ask what type of workflow will be used by the workspace. You will be presented with three options, CLI, VCS, and API.

The screenshot shows the initial steps of creating a new workspace in Terraform Cloud:

- 1 Choose Type**: The first step, currently selected.
- 2 Connect to VCS**: The second step.
- 3 Choose a repository**: The third step.
- 4 Confirm changes**: The fourth step.

**Choose your workflow**

- Version control workflow** (Most common):  
Description: Store your Terraform configuration in a git repository, and trigger runs based on pull requests and merges.  
Learn More
- CLI-driven workflow**:  
Description: Trigger remote Terraform runs from your local command line.  
Learn More
- API-driven workflow**:  
Description: A more advanced option. Integrate Terraform into a larger pipeline using the Terraform API.  
Learn More

**Figure 8:** Terraform Cloud Workspace - Notifications

We have been operating within the CLI workflow up to this point in the course, triggering all commands from the command line. You control the Terraform workflow from the CLI with standard Terraform commands like `plan` and `apply`.

The VCS or version control system workflow is the most common option, but it also requires that you host the Terraform code in a VCS repository. Events on the repository will trigger workflows on Terraform Cloud. For instance, a commit to the default branch could kick off a `plan` and `apply` workflow in Terraform Cloud.

If you need more customized automation and workflows than what is available in Terraform Cloud, you can use an API workflow to integrate Terraform Cloud into a larger automation pipeline. For instance, if you are already using Jenkins or CI/CD Pipelines to automate your IaC deployment, you can hook





Terraform Cloud in to handle the Terraform actions.

## Task 2: Create a Terraform Cloud Workspace for DEV deployments

A workspace name has to be all lower case letters, numbers, and dashes. The recommended naming convention from HashiCorp is the team name, the cloud the infrastructure will be deployed in, the application or purpose of the infrastructure, and the environment, whether it's dev, staging, prod, etc. Workspace names are relative to the organization, so they do not have to be globally unique, only unique within the organization.

Suggested workspace naming convention —, e.g. devops-aws-myapp-dev which makes it easier to filter, navigate and performed automated operations against Terraform Cloud workspaces.

Let's create a new development workspace for your app called: [devops-aws-myapp-dev](#)

1. Select [New Workspace](#) and choose the [CLI-driven workflow](#)
2. Give the workspace a name of [devops-aws-myapp-dev](#)
3. Select [Create Workspace](#)

## Task 3: Create a Terraform Cloud Workspace for PROD deployments

The state data in a workspace can also be shared with other workspaces in the organization as a data source. That's helpful if you decide to refactor your Terraform code into separate workspaces, but you need to pass information between them.

To isolate our production deployment from our development, let's create a new workspace for our production infrastructure named: [devops-aws-myapp-prod](#)

1. Select [New Workspace](#) and choose the [CLI-driven workflow](#)
2. Give the workspace a name of [devops-aws-myapp-prod](#)
3. Select [Create Workspace](#)

## Task 4: Assign and set variables per workspace

### 3.1 - Assign Variables to DEV workspace

- Navigate to your Terraform Cloud [devops-aws-myapp-dev](#) workspace
- Once there, navigate to the [Variables](#) tab.





- In the [Variables](#) tab, you can add variables related to the state file that was previously migrated.
- To do this, first select the [+ Add variable](#) button
- Let's add a Terraform variable named `aws_region` with a value of `us-east-1`
- Let's add a second Terraform variable named `vpc_name` with a value of `dev_vpc`
- Let's add a third Terraform variable named `environment` with a value of `dev`
- Choose the [Apply variable set](#) option to apply the [AWS Credentials](#) variable set

### 3.2 - Assign Variables to PROD workspace

- Navigate to your Terraform Cloud `devops-aws-myapp-prod` workspace
- Once there, navigate to the [Variables](#) tab.
- In the [Variables](#) tab, you can add variables related to the state file that was previously migrated.
- To do this, first select the [+ Add variable](#) button
- Let's add a Terraform variable named `aws_region` with a value of `us-east-1`
- Let's add a second Terraform variable named `vpc_name` with a value of `prod_vpc`
- Let's add a third Terraform variable named `environment` with a value of `prod`
- Choose the [Apply variable set](#) option to apply the [AWS Credentials](#) variable set

## Task 5: Change Terraform versions per workspace

Modify the Terraform version for the development workspace to use `1.1.2` to test if the deployments are compatible with that release and set production workspace at terraform version `1.0.11`

To set the Terraform version for a workspace navigate to [Settings > General](#) of the workspace and specify the appropriate Terraform core version.

## Task 6: Deploy infrastructure using Terraform Cloud workspaces

### 6.1 - Declare Environment variables

The last step is to now deploy our infrastructure into the appropriate workspace. Now that our infrastructure will be broken up by environment, let's declare a variable in our `variables.tf` named `environment`

`variables.tf`

```
variable "environment" {
```





```

  type = string
  description = "Infrastructure environment. eg. dev, prod, etc"
  default = "test"
}

```

Let's also update a couple of our resource blocks inside the `main.tf` to make use of this new environment variable.

`main.tf`

```

resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name      = var.vpc_name
    Environment = var.environment
    Terraform  = "true"
  }

  enable_dns_hostnames = true
}

resource "aws_key_pair" "generated" {
  key_name    = "MyAWSKey${var.environment}"
  public_key  = tls_private_key.generated.public_key_openssh
}

```

## 6.2 - Deploy infrastructure using appropriate Terraform Cloud workspaces

Since we have Terraform Cloud workspaces that share the same Terraform codebase we will leverage backend partial configuration to select the correct workspace. If you are unfamiliar with Terraform backend partial configuration it is recommended you review the Terraform Backend Configuration lab which explains it in detail.

Create two new files in our working directory called `dev.hcl` and `prod.hcl`

`dev.hcl`

```
workspaces { name="devops-aws-myapp-dev"}
```

`prod.hcl`

```
workspaces { name="devops-aws-myapp-prod"}
```

Initialize and apply configuration in the development workspace.





```
terraform init -backend-config=dev.hcl -reconfigure
terraform plan
terraform apply
```

Initialize and apply configuration in the production workspace.

```
terraform init -backend-config=prod.hcl -reconfigure
terraform plan
terraform apply
```

You can view each of these deployments from either the CLI or Terraform Cloud interface. The infrastructure can be validated by looking at the Terraform Cloud workspace dashboard.

Workspaces 4 total			
WORKSPACE NAME	RUN STATUS	REPO	LATEST CHANGE
devops-aws-myapp-dev	✓ Applied		2 minutes ago
devops-aws-myapp-prod	⟳ Applying		34 minutes ago
getting-started	✓ Applied		a month ago
my-aws-app	✓ Applied		3 hours ago

**Figure 9:** Terraform Cloud Workspace - Dashboard

### 6.3 - Destroy infrastructure in appropriate workspace

If you would like to cleanup and destroy your infrastructure to keep costs down, that can be done at the workspace level. Navigate to the [Settings](#) of the workspace and select [Destruction and Deletion](#). That will provide you with the ability to [Queue destroy plan](#) which follows the same workflow as a [terraform destroy](#). Be sure not to delete the workspace itself (which is also an option on this same page) as in a future lab we will connect these new workspaces to the VCS workflow trigger runs based on code commits rather than via CLI.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

devops-aws-myapp-dev

No workspace description available. [Add workspace description](#).

Overview Runs States Variables **Settings** Actions

Resources 30 Terraform version 1.1.2 Updated 7 minutes ago

**Destruction and Deletion**

There are two independent steps for destroying this workspace and any infrastructure associated with it. First, any Terraform infrastructure should be destroyed. Second, the workspace in Terraform Cloud, including any variables, settings, and alert history can be deleted.

**Destroy infrastructure**

**Allow destroy plans**  
When enabled, this setting allows a destroy plan to be created and applied. This also applies when using the CLI.

**Manually destroy**

Queuing a destroy plan will redirect to a new plan that will destroy all of the infrastructure managed by Terraform. It is equivalent to running `terraform plan --destroy -out=destroy.tfplan` followed by `terraform apply destroy.tfplan` locally.

**Queue destroy plan** ←

**Delete Workspace**

Deleting a workspace will remove any variables, settings, alert history, run history, and state related to it. This **will not** remove any infrastructure managed by this workspace. If needed, destroy the infrastructure prior to deleting the workspace.

**Delete from Terraform Cloud**

**Figure 10:** Terraform Cloud Workspace - Dashboard

## What belongs in a workspace?

Often times we are asked - What should I put in a workspace? We recommend infrastructure that should be managed together as a unit be placed into the same workspace. Who has to manage it, how often does it change, does it have external dependencies that we can't control. Ask these questions. Think about what happens when you run `terraform apply`. You should be able to describe what you just built, and what outputs it provides, who this infrastructure is for, and how to utilize it.

A workspace could be: An entire application stack from network on up. Great for dev environments that you want to be completely self-contained. Or it could be a workspace that builds core network infrastructure and nothing else. Maybe the network team has to manage that. They get to be the lords of the network and provide terraform outputs to those who need a VPC or subnet. You can also use workspaces to deploy platform services like Kubernetes.





## Lab: Terraform Cloud Secure Variables

Terraform Cloud has built in support for encryption and storage of variables used within your Terraform configuration. This allows you to centrally manage variables per workspace or organization as well as store sensitive items (such as cloud credentials, passwords, etc.) securely during the provisioning process without exposing them in plaintext or storing them on someone's laptop.

This lab demonstrates how to store variables to Terraform Cloud.

- Task 1: Utilize Terraform Cloud Variables
- Task 2: Run a plan locally
- Task 3: Change a variable
- Task 4: Run an apply remotely
- Task 5: Utilize Variable sets

This lab demonstrates how to store variables in Terraform Cloud.

### Task 1: Utilize Terraform Cloud Variables

Now that we have our state stored in Terraform Cloud in our workspace, we will take the next logical step and store our variables in Terraform Cloud

#### Step 1.1 - Create Terraform Variable

Note: If your infrastructure has not yet been deployed, execute a `terraform apply` to deploy the infrastructure.

- Navigate to your Terraform Cloud `my-aws-app` in the UI.
- Once there, navigate to the `Variables` tab.
- In the `Variables` tab, you can add variables related to the state file that was previously migrated.
- To do this, first select the `+ Add variable` button
- Let's add a Terraform variable named `aws_region` with a value of `us-east-1`
- Let's add a second Terraform variable named `vpc_name` with a value of `demo_vpc`

#### Step 1.2 - Create Environment Variable

Terraform requires credentials in order to communicate with your cloud provider's API. These API keys should never be stored directly in your terraform code. We will use Terraform Cloud environment





variables to store our sensitive cloud credentials for AWS.

- In the [Variables](#) tab of your [my-aws-app](#) you can add environment variables for the AWS Credentials
- To do this, first select the [+ Add variable](#) button
- Lets add a Environment variable named [AWS\\_ACCESS\\_KEY\\_ID](#) with your AWS Access Key
- Lets add a second Environment variable named [AWS\\_SECRET\\_ACCESS\\_KEY](#) with your AWS Secret Key. Be sure to mark this variable as sensitive. Sensitive variables will not be displayed within the environment, and can only be overwritten - not read.

## Task 2: Run a plan locally

### Step 2.1 - Run a terraform init

- Next reinitialize your terraform project locally and run a [terraform plan](#) to validate that refactoring your code to make use of variables within TFC did not introduce any planned infrastructure changes. This can be confirmed by validating a zero change plan.

```
terraform plan

Running plan in the remote backend. Output will stream here. Pressing Ctrl
-C
will stop streaming the logs, but will not stop the plan running remotely.

Preparing the remote plan...

To view this run in a browser, visit:
https://app.terraform.io/app/Enterprise-Cloud/server-build/runs/run-DGXauYrWeB1xwwPx

Waiting for the plan to start...

Terraform v0.14.8
Configuring remote state backend...
Initializing Terraform configuration...
module.keypair.tls_private_key.generated: Refreshing state... [id=34
    a0559a16dc68108d30a76d9a5a7b25f8885e1e]
module.keypair.local_file.public_key_openssh[0]: Refreshing state... [id=0
    e346a51831a9bc96fd9ea142f8c35b4e1ade12b]
module.keypair.local_file.private_key_pem[0]: Refreshing state... [id=
    b7ac3f7125c4e3681fd38539b220c1abf01d1254]
module.keypair.null_resource.chmod[0]: Refreshing state... [id
    =1854006565944356631]
```





```
module.keypair.aws_key_pair.generated: Refreshing state... [id=terraform-nyl-ant-key]
module.server[0].aws_instance.web[1]: Refreshing state... [id=i-047d4d406e22e87f9]
module.server[1].aws_instance.web[0]: Refreshing state... [id=i-003fcdb877e575b26]
module.server[0].aws_instance.web[0]: Refreshing state... [id=i-0d16b6f6eda6b834e]
module.server[1].aws_instance.web[1]: Refreshing state... [id=i-03be3bf6ee29f5633]
```

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, no actions need to be performed.

### Task 3: Change a Variable

Next, we will be testing to make sure that our variables in TFC are working correctly.

#### Step 3.1 - Modify the `vpc` name by changing the `vpc_name` variable in TFC

- Simply click on the \*\*\* button to the right of your variables, and then select the `Edit` option
- Change the value of your `vpc_name` variable and save to apply a new name for your VPC.
- Run a `terraform apply` locally in your code editor.
- When this is ran locally, you should see that Terraform is looking at the variable in TFC, and it will update your server tags with the `vpc_name` value.

### Task 4: Run an apply remotely

Now that we've tested that our configuration is still working and that our variables can be stored and manipulated within Terraform Cloud, let's also test an apply within Terraform Cloud itself.

#### Step 4.1 - Queue a plan in Terraform Cloud

Now that your variable value has changed, let's queue the plan in Terraform Cloud

- Navigate to the `Actions` box in the top right corner and choose `Start new plan`





- In the `Reason for starting plan` dialogue box, type something meaningful to your use case. For this, we can simply say `Updating VPC Name`.
- In the `Choose plan type` option, make sure that `Plan (most common)` is selected
- Follow the dialogue boxes through this process. Review your plan to ensure that it will be doing exactly what you expect, and then confirm and apply when given the opportunity.

### Task 5: Utilize Variable sets

Variable sets allow Terraform Cloud users to reuse both Terraform-defined and environment variables not just from root to child modules, but across certain workspaces or an entire organization. One of the most common use cases for variable sets is credential and identity management. Variable sets allow for a convenient way to reuse variables across workspaces within a Terraform Cloud organization.

- In the `Settings` tab of your Terraform Cloud Organization go to `Variable sets`.
- Select the `Create Variable set` button
- Give your variable set a name - `AWS Credentials` and description.
- Apply the variable set to your `my-aws-app` workspace.
- You can add environment variables for the AWS Credentials
- To do this, first select the `+ Add variable` button
- Lets add a Environment variable named `AWS_ACCESS_KEY_ID` with your AWS Access Key
- Lets add a second Environment variable named `AWS_SECRET_ACCESS_KEY` with your AWS Secret Key. Be sure to mark this variable as sensitive.

Now you can use these AWS credentials on any new workspace you create in Terraform Cloud by adding the variable set to the workspace. While storing cloud credentials is a common use of variable sets, they are not limited to credentials. Variable sets can be utilized for any set of variables that you want to reuse across workspaces within the organization.

**Congratulations! You're now storing your variable definitions remotely and can control the behavior of your Terraform code by adjusting their values. With Terraform Cloud you are able to centralize and secure the variable definitions for your workspace.**





## Lab: Terraform Cloud - Version Control

In order for different teams and individuals to be able to work on the same Terraform code, you need to use a Version Control System (VCS). Terraform Cloud can integrate with the most popular VCS systems including GitHub, GitLab, Bitbucket and Azure DevOps.

This lab demonstrates how to connect Terraform Cloud to your personal GitHub account.

- Task 1: Sign-up/Sign-in to GitHub
- Task 2: Create Terraform Cloud VCS Connection
- Task 3: Verify Connection

### Task 1: Configuring GitHub Access

You will need a free GitHub.com account for this lab. We recommend using a personal account. You can sign up or login in to an existing account at <https://github.com/>

### Task 2: Create Terraform Cloud VCS Connection

1. Login to github in one browser tab.
2. Login to Terraform Cloud in another browser tab.
3. Within Terraform Cloud navigate to the settings page

The screenshot shows the Terraform Cloud interface. At the top, there's a dark blue header with the HashiCorp logo, the user name 'trainer-admin' with a dropdown arrow, and navigation tabs for 'Workspaces', 'Modules', and 'Settings'. The 'Settings' tab is highlighted in white. Below the header, a breadcrumb trail reads 'trainer-admin / Settings / Profile'. The main content area is currently empty, indicated by a large light gray box.

4. Click “VCS Providers” link:

The screenshot shows the 'VCS Providers' page. At the top, there's a dark blue header with the HashiCorp logo, the user name 'trainer-admin' with a dropdown arrow, and navigation tabs for 'Workspaces', 'Modules', and 'Settings'. The 'Settings' tab is highlighted in white. Below the header, a breadcrumb trail reads 'trainer-admin / Settings / VCS Providers'. The main content area contains a table with one row, labeled 'ORGANIZATION SETTINGS'. To the right of the table is a blue button labeled 'Add a VCS Provider'.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



For full instructions follow the **Configuring GitHub Access** section of the Terraform Cloud documentation to connect your GitHub account to your Terraform Organization.

Configuring GitHub Access - <https://www.terraform.io/docs/cloud/vcs/github.html>

Note: The process involves several back and forth changes and is documented well in the link.

### Task 3: Verify Connection

1. Navigate to <https://app.terraform.io> and click “+ New Workspace”
2. Click the VCS Connection in the “Source” section.
3. Verify you can see repositories:

### Create a new Workspace

New workspace   Import from legacy (Atlas) environment

This workspace will be created under the current organization, **trainer-admin**.

**WORKSPACE NAME**  
e.g. workspace-name

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about [naming workspaces](#).

**SOURCE**  
None   GitHub   +

**REPOSITORY**  
e.g. organization/repository-name

- azure-terraform-workshop/azureworkshop-workspaces
- azure-terraform-workshop/terraform-workshop-labs
- azure-terraform-workshop/terraform-azurerm-appserver
- azure-terraform-workshop/terraform-azurerm-dataserver
- azure-terraform-workshop/terraform-azurerm-networking

If you can see repositories then you are all set.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



In future labs we will utilize the VCS connection to connect our Terraform code to Terraform Cloud workspaces as well as the Private Module Registry.





## Lab: Terraform Cloud - Private Module Registry

Terraform Cloud's Private Module Registry allows you to store and version Terraform modules which are re-usable snippets of Terraform code. It is very similar to the Terraform Public Module registry including support for module versioning along with a searchable and filterable list of available modules for quickly deploying common infrastructure configurations.

The Terraform Private Module Registry is an index of private modules that you don't want to share publicly. This lab demonstrates how to register a module with your Private Module Registry then reference it in a workspace.

- Task 1: Move terraform module code to a private GitHub repository
- Task 2: Publish module to Private Module Registry
- Task 3: Review module block reference to Private Module Registry

### Task 1: Move terraform module code to a private GitHub repository

Instead of writing a terraform module from scratch we will copy an existing S3 module from the public Terraform registry. Visit this URL to view the AWS S3 module:

<https://github.com/terraform-aws-modules/terraform-aws-s3-bucket>

#### Step 1.1 Fork the Module Repository to your GitHub account

You are going to fork the following repository into your own GitHub account:

- <https://github.com/terraform-aws-modules/terraform-aws-s3-bucket>

This repository represents a module that can be developed and versioned independently. Note the **Source:** link that points at the github repository for this module. Click on the Source URL and create your own fork of this repository with the **Fork** button.

### Task 2: Publish module to Private Module Registry

We need to add this repository into the Private Module Registry. Navigate back to Terraform Cloud and click the "Modules" menu at the top of the page. From there click the "+ Add Module" button.





## Add Module

This module will be created under the current organization, **example-org-6cde13**. Modules can be added from all [supported VCS providers](#).

1 Connect to VCS

2 Choose a repository

3 Confirm selection

### Choose a repository

Choose the repository that hosts your module source code. We'll watch this for commits and tags. The format of your repository name should be `terraform-<PROVIDER>-<NAME>`.

12 repositories  Filter

gmaentz/terraform-aws-s3-bucket	>
gmaentz/terraform-aws-server	>
gmaentz/terraform-example-module	>

Select the S3 repository you forked earlier.

## Add Module

This module will be created under the current organization, **example-org-6cde13**. Modules can be added from all [supported VCS providers](#).

1 Connect to VCS

2 Choose a repository

3 Confirm selection

### Confirm selection

Provider GitHub (GitHub.com)

Repository gmaentz/terraform-aws-s3-bucket

[Publish module](#)

[Cancel](#)



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



Note: You will see your github user name since you forked this repo.

Click “Publish Module”.

This will query the repository for necessary files and tags used for versioning.

Private

### s3-bucket

Terraform module which creates S3 bucket resources on AWS

Published by example-org-6cde13   Provider **aws** aws

Version Published Provisions Source  
1.17.0 a few seconds ago < 100 gmaentz/terraform-aws-s3-bucket

Submodules Examples

Readme Inputs (25) Outputs (8) Dependencies (0) Resources (7)

#### AWS S3 bucket Terraform module

Terraform module which creates S3 bucket on AWS with all (or almost all) features provided by Terraform AWS provider.

This type of resources are supported:

- S3 Bucket
- S3 Bucket Policy
- S3 Bucket Notification - use [modules/notification](#) to configure notifications to Lambda functions, SQS

Manage Module for Organization

Open in Designer

#### Usage Instructions

Copy and paste into your Terraform configuration and set values for the input variables. Or, [design a configuration](#) to easily use module and workspace outputs as inputs.

#### Copy configuration details

```
module "s3-bucket" {  
  source = "app.terraform.io/example-org-6cde13/s3-bucket/aws"  
  version = "1.17.0"  
  # insert required variables here  
}
```

When running Terraform on the CLI, you must [configure credentials in .terraformrc or terraformrc](#) to access this module.

### Task 3: Review module block reference to Private Module Registry

Now that the module has been published to the Private Module registry, we can utilize it within any module block in our terraform configuration. Note the `source` and `version` of the newly published private module.

```
module "s3-bucket" {  
  source = "app.terraform.io/example-org-6cde13/s3-bucket/aws"  
  version = "1.17.0"  
  # insert required variables here  
}
```

### Resources

- Private Registries
- Publishing Modules

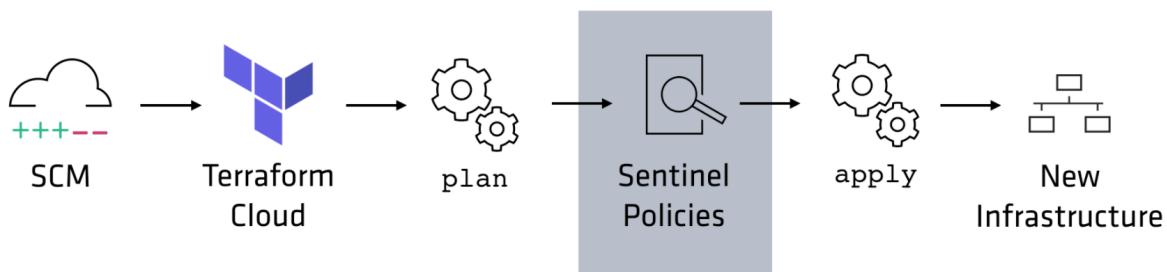




## Lab: Terraform Cloud - Sentinel Policy

Sentinel is the Policy-as-Code product from HashiCorp that automatically enforces logic-based policy decisions across all HashiCorp Enterprise products. It allows users to implement policy-as-code in a similar way to how Terraform implements infrastructure-as-code. If enabled, Sentinel is run between the terraform plan and apply stages of the workflow.

## How it works with Sentinel



**Figure 1:** Sentinel Workflow

In this lab we are going to use sentinel policy within Terraform Cloud to determine if it should deploy the infrastructure from our plan based on policy enforcement.

- Task 1: Setup Workspace in Terraform Cloud
- Task 2: Create Sentinel Policy
- Task 3: Apply policy sets to the organization and application workspace
- Task 4: Test and Enforce Sentinel Policy

### Task 1: Setup Workspace in Terraform Cloud

You should already have a Terraform Cloud account with Team and Governance capabilities, if it has been less 30 days since you activated the trial on your Terraform Cloud account you are good to go. If you do not have a Terraform Cloud account already, please create an account.





## Task 2: Create Sentinel Policy

Let's write a Sentinel policy to set guardrails around one of type of resources created by our Terraform configuration. We want to enforce two organization requirements for EC2:

- EC2 instances must have a Name tag.
- EC2 instances must be of type t2.micro, t2.small, or t2.medium.

First we need a place to store our policies. Sentinel is a policy as code framework, so just like Terraform code we should store Sentinel code in a Version Control System like GitHub.

Create a fork of the following GitHub repo, which contains several Sentinel policies that you can use in your own organization. Use the **Fork** button in the upper right corner of the screen to create your own copy into your GitHub account.

<https://github.com/btkrausen/hashicorp>

### 2.1 Sentinel Policies

The Sentinel policies we will be using from this forked repo are:

- `enforce-mandatory-tags` policy.
- `restrict-ec2-instance-type` policy.

The `enforce-mandatory-tags` policy requires all EC2 instances to have a `Name` tag and the `restrict-ec2-instance-type` policy only allows EC2 instances of type: `t2.micro`, `t2.small`, or `t2.medium`.

These policies use some functions from a Sentinel module in a different repository called `terraform-guides`. You'll find many useful Sentinel policies and functions in its governance/third-generation directory.

### 2.1 Sentinel Policy Set

You can group individual Sentinel policies together and specify their enforcement level using a Sentinel policy set. We will be using a Sentinel policy set from the forked repo which is contained in a `sentinel.hcl` file. A policy set is a collection of sentinel policies are defined within the `sentinel.hcl` file. This level of enforcement for your Sentinel policies is also defined here. If you have multiple policies in your policy repository, you list them within a policy set and Terraform Cloud applies the policies in the





order they appear in this file. We are also making use of sentinel modules to include some functions used by our Sentinel policies.

### sentinel.hcl

```
module "tfplan-functions" {
  source = "https://raw.githubusercontent.com/hashicorp/terraform-guides/
    master/governance/third-generation/common-functions/tfplan-functions/
    tfplan-functions.sentinel"
}

module "tfconfig-functions" {
  source = "https://raw.githubusercontent.com/hashicorp/terraform-guides/
    master/governance/third-generation/common-functions/tfconfig-
    functions/tfconfig-functions.sentinel"
}

policy "enforce-mandatory-tags" {
  enforcement_level = "advisory"
}

policy "restrict-ec2-instance-type" {
  enforcement_level = "hard-mandatory"
}
```

Policy enforcement levels are also defined inside the policy set. Sentinel has three enforcement levels:

- **Advisory:** The policy is allowed to fail. However, a warning should be shown to the user or logged.
- **Soft Mandatory:** The policy must pass unless an override is specified. The semantics of “override” are specific to each Sentinel-enabled application. The purpose of this level is to provide a level of privilege separation for a behavior. Additionally, the override provides non-repudiation since at least the primary actor was explicitly overriding a failed policy.
- **Hard Mandatory:** The policy must pass no matter what. The only way to override a hard mandatory policy is to explicitly remove the policy. Hard mandatory is the default enforcement level. It should be used in situations where an override is not possible.

### Task 3: Apply policy sets to the Terraform Cloud organization

Now that the policies and policy sets are defined, let's connect them to our Terraform Cloud organization.

1. Go into the **Organization Settings** for your training organization and click on **Policy Sets**.





## Policy Sets

[Create a new policy set](#)

Policy sets let you group policies into categories and enforce them on specific workspaces.

No policy sets have been created for this organization.

2. Use the **Connect a new policy set** button to connect your new GitHub repo to your organization. Remember, the repository is named **hashicorp**.
3. Under **Name** enter “AWS-Global-Policies”
4. Under **Description** you can enter “Sentinel Policies for our AWS resources”.
5. In the **More Options** menu set the **Policies Path** to `terraform/Lab Prerequisites/Terraform Cloud Sentinel/global`. This tells Terraform Cloud to use the AWS specific policies that are stored in the repo.
6. Leave everything else at its default setting and click on the **Connect policy set** button at the bottom of the page.

## Task 4: Test and Enforce Sentinel Policy

We can test our Sentinel policies by performing a Terraform run on our Terraform Cloud workspaces.

### 4.1 Manually Run a Plan

Navigate to your “my-aws-app” and queue a plan.

### 4.2 Review the Plan

You will now see a new phase within the Terraform run called Policy Check. We see that the plan was successful with both policy checks passing.



# HashiCorp Certified: Terraform Associate



## Hands-On Labs

my-aws-app

No workspace description available. [Add workspace description](#).

Resources 0      Terraform version 1.0.11      Updated a few seconds ago

Overview    **Runs**    States    Variables    Settings    Actions

### ! Policy checked Build out Terraform Configuration CURRENT

- gabe\_maentz triggered a run from UI a few seconds ago
- Plan finished a few seconds ago
- Cost estimation finished a few seconds ago
- Policy check passed a few seconds ago
- Apply pending

Needs Confirmation: Check the plan and confirm to apply it, or discard the run.

**Figure 2:** Sentinel Policy Check

You can navigate into the Policy check portion of the run to see more details about the policies included in the policy set and their status





✓ Policy check passed 3 minutes ago Policies: 2 passed, 0 failed ^

Queued 3 minutes ago > Passed 2 minutes ago

✓ passed AWS-Global-Policies/enforce-mandatory-tags  
✓ passed AWS-Global-Policies/restrict-ec2-instance-type

[View Logs](#) [View JSON Data](#)

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

```
===== Results for policy set: AWS-Global-Policies =====
Sentinel Result: true
This result means that all Sentinel policies passed and the protected behavior is allowed.
2 policies evaluated.
## Policy 1: AWS-Global-Policies/enforce-mandatory-tags (advisory)
Result: true
Description:
  This policy uses the Sentinel tfplan import to require that all EC2 instances have all mandatory tags. Note that the comparison is case-sensitive since ./enforce-mandatory-tags.sentinel:24:1 - Rule "main"
  Description:
    Main rule
```

**Figure 3:** Sentinel Policy Check - Passed

Confirm & Apply the plan.

#### 4.3 Update the Terraform Code to update the instance size

Now let's update our code to specify an instance size that is not allowed via policy. Navigate to the `web_server` resource block and update the `instance_type` to `m5.large`

```
resource "aws_instance" "web_server" {
  ami                               = data.aws_ami.ubuntu.id
  instance_type                     = "m5.large"
  subnet_id                         = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups                   = [aws_security_group.vpc-ping.id,
                                         aws_security_group.ingress-ssh.id, aws_security_group.vpc-web.id]
  associate_public_ip_address      = true
  key_name                           = aws_key_pair.generated.key_name
  connection {
    user     = "ubuntu"
    private_key = tls_private_key.generated.private_key_pem
    host     = self.public_ip
  }
}
```





#### 4.4 Manually Run a Plan

Queue a new terraform run that includes this change by initiating a `terraform init` and `terraform plan`

```
terraform init  
terraform plan
```

#### 4.5 Review the Plan

Will see the plan was unsuccessful because of a policy failure in moving to an `instance_type` that is not allowed.

`Result: false`

`Description:`

This policy uses the Sentinel tfplan/v2 `import` to require that all EC2 instances have instance types from an allowed list

`Print messages:`

```
aws_instance.web_server has instance_type with value m5.large that is not  
in the allowed list: [t2.micro, t2.small, t2.medium]
```

The policy is set to `hard-mandatory` enforcement that prevents the plan from moving forward. The enforcement level can be adjusted within the policy set `sentinel.hcl` if desired.





**✖ Policy check hard failed** a few seconds ago

Policies: 1 passed, 1 hard failed ▾

Queued a few seconds ago > Hard failed a few seconds ago

passed AWS-Global-Policies/enforce-mandatory-tags  
failed AWS-Global-Policies/restrict-ec2-instance-type

[View Logs](#) [View JSON Data](#)

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

```
===== Results for policy set: AWS-Global-Policies =====
Sentinel Result: false

This result means that one or more Sentinel policies failed. More than likely,
this was due to the discovery of violations by the main rule and other
sub-rules. Please see the details of the policies executed below to find the
violation(s), which is usually indicated by a rule with a false boolean value,
or non-zero collection data.

2 policies evaluated.

## Policy 1: AWS-Global-Policies/restrict-ec2-instance-type (hard-mandatory)

Result: false

Description:
  This policy uses the Sentinel tfplan/v2 import to require that all EC2 instances have instance types from an allowed list

Print messages:
```

**Figure 4:** Sentinel Policy Check - Failed

#### Step 4.6 Update the configuration to maintain compliance

Now let's update our code to specify an instance size that is allowed via policy. Navigate to the `web_server` resource block and update the `instance_type` to `t2.medium`

```
resource "aws_instance" "web_server" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type                = "t2.medium"
  subnet_id                    = aws_subnet.public_subnets["public_subnet_1"]
  ".id"
  security_groups              = [aws_security_group.vpc-ping.id,
    aws_security_group.ingress-ssh.id, aws_security_group.vpc-web.id]
  associate_public_ip_address = true
  key_name                     = aws_key_pair.generated.key_name
  connection {
    user           = "ubuntu"
    private_key   = tls_private_key.generated.private_key_pem
    host          = self.public_ip
  }
}
```

Queue a new terraform run:



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



### terraform plan

This time the planned change is highlighted, along with the cost impact and is compliant with our organizational policy.

✓ Plan finished a minute ago

Started a minute ago > Finished a few seconds ago

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

Terraform will perform the following actions:

```
# aws_instance.web_server will be updated in-place
~ resource "aws_instance" "web_server" {
  id                      = "i-09d179b13aeeef92f3"
  ~ instance_type          = "t2.micro" -> "t2.medium"
  tags                    =
    "Name" = "Web EC2 Server"
}
# (28 unchanged attributes hidden)

}
# (5 unchanged blocks hidden)

Plan: 0 to add, 1 to change, 0 to destroy.
```

[Download Sentinel mocks](#) [Sentinel mocks can be used for testing your Sentinel policies](#)

✓ Cost estimation finished a minute ago

Resources: 3 of 6 estimated · \$54.18/mo · +\$25.89

Policies: 2 passed, 0 failed

✓ Policy check passed a minute ago

Queued a few seconds ago > Passed a few seconds ago

passed AWS-Global-Policies/enforce-mandatory-tags  
passed AWS-Global-Policies/restrict-ec2-instance-type

[View Logs](#) [View JSON Data](#) [View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

```
=====
Results for policy set: AWS-Global-Policies =====
Sentinel Result: true
This result means that all Sentinel policies passed and the protected behavior is allowed.
2 policies evaluated.
## Policy 1: AWS-Global-Policies/enforce-mandatory-tags (advisory)
Result: true
Description:
  This policy uses the Sentinel tfplan import to require that all EC2 instances have all mandatory tags. Note that the comparison is case-sensitive since AWS tags are case-sensitive.
./enforce-mandatory-tags.sentinel:24:1 - Rule "main"
```

**Figure 5:** Sentinel Policy Check - End-to-End Run

## Resources

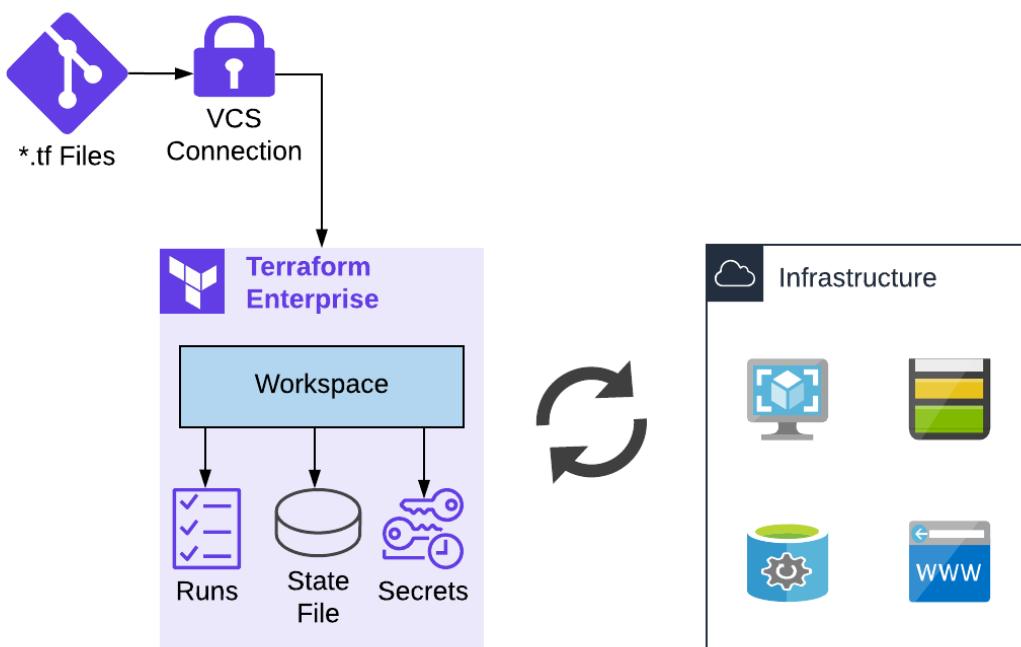
- [Sentinel Language Spec](#)
- [Experiment in the Sentinel Playgroun](#)d





## Lab: Terraform Cloud - Version Control Workflow

Once multiple people are collaborating on Terraform configuration, new steps must be added to the core Terraform workflow (Write, Plan, Apply) to ensure everyone is working together smoothly. In order for different teams and individuals to be able to work on the same Terraform code, you need to use a Version Control System (VCS). The Terraform Cloud VCS or version control system workflow includes the most common steps necessary to work in a collaborative nature, but it also requires that you host the Terraform code in a VCS repository. Events on the repository will trigger workflows on Terraform Cloud. For instance, a commit to the default branch could kick off a plan and apply workflow in Terraform Cloud.



**Figure 1:** Terraform Cloud/Enterprise - Version Control Workflow

Terraform Cloud can integrate with the most popular VCS systems including GitHub, GitLab, Bitbucket and Azure DevOps. This lab demonstrates using the Terraform Cloud VCS workflow and its native integrations with GitHub.

- Task 1: Create a new GitHub repository
- Task 2: Update your Repository and Commit Changes to GitHub





- Task 3: Migrate to the VCS workflow
- Task 4: Validate variables for each Terraform Cloud Workspace
- Task 5: Update TFC development workspace to use development branch
- Task 6: Use the VCS workflow to deploy an update to development
- Task 7: Use the VCS workflow to merge an update to production

## Task 1: Create a new GitHub repository

You will need a free GitHub.com account for this lab. We recommend using a personal account that was configured in previous labs. You can sign up or login in to an existing account at <https://github.com/>

Login to github and create a new repository by navigating to <https://github.com/new>

Use the following settings for the code repository

- Name: “my-app”
- Description: My AWS Application
- Private repo

Once created, connect the the repository to your local machine.

```
git init
git remote add origin https://github.com/<YOUR_GIT_HUB_ACCOUNT>/my-app.git
```

## Task 2: Update your Repository and Commit Changes to GitHub

In your repository create a `.gitignore` file to filter out any items we don’t wish to check into version control at this time.

`.gitignore`

```
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# Crash log files
crash.log

# Ignore any .tfvars files that are generated automatically for each
Terraform run. Most
```





```
# .tfvars files are managed as part of configuration and so should be
# included in
# version control.
#
# example.tfvars

# Ignore override files as they are usually used to override resources
# locally and so
# are not checked in
override.tf
override.tf.json
*_override.tf
*_override.tf.json

# Include override files you do wish to add to version control using
# negated pattern
#
# !example_override.tf

# Include tfplan files to ignore the plan output of command: terraform
# plan -out=tfplan
# example: *tfplan*
.DS_Store
tfc-getting-started/
*.hcl
terraform.tf
```

## README.md

```
# My App
```

In **this** repo, you'll find a quick and easy app to get started using [[Terraform Cloud](https://app.terraform.io/)] (<https://app.terraform.io/>) with [[GitHub](https://github.com/)] (<https://github.com/>).

### ## Version Control Workflow

Once multiple people are collaborating on Terraform configuration, new steps must be added to the core Terraform workflow (Write, Plan, Apply) to ensure everyone is working together smoothly. In order for different teams and individuals to be able to work on the same Terraform code, you need to use a Version Control System (VCS). The Terraform Cloud VCS or version control system workflow includes the most common steps necessary to work in a collaborative nature, but it also requires that you host the Terraform code in a VCS repository. Events on the repository will trigger workflows on Terraform Cloud. For instance, a commit to the default branch could kick off a plan and apply workflow in Terraform Cloud.





```
## Master Terraform by taking a Hands-On Approach
```

Check out the 70+ labs that follow the HashiCorp Certified: Terraform Associate certification. Learn more at <https://www.udemy.com/course/terraform-hands-on-labs>

Commit the changes in GitHub.

```
git add .
git commit -m "terraform code update for my app"
git push --set-upstream origin master
```

### Task 3: Migrate to the VCS workflow

#### Connect Terraform Cloud Workspace to the GitHub my-app repository

1. Navigate to <https://app.terraform.io> and click the `devops-aws-myapp-prod` workspace
2. Select Settings » Version Control
3. Click the “Connect to version control”
4. Select the Version Control workflow
5. Click the VCS Connection in the “Source” section.
6. Verify you can see repositories and select the `my-app` github repository.

You can see that we are connected to our GitHub my-app project that we reviewed earlier. The `VCS branch` will denote which branch in GitHub that this workspace will be watching for changes. Since our triggering is set to `Always trigger runs`, any changes to our my-app project in the master branch of GitHub will trigger this workspace to run.

7. Select Update VCS Settings
8. Validate that a new Terraform run will occur on the workspace. Confirm and Apply the Terraform run.

### Task 4: Validate variables for each Terraform Cloud Workspace

Version control allows us to store our Terraform configuration in a centralized location and reuse the same code base across branches within our code repository. We however may wish to change a few specific settings to distinguish our development and production deployments. We will utilize Terraform variables within Terraform Cloud to specify these unique environment settings.





Update the variables for each of the `myapp` Terraform Cloud workspaces for the `environment` variable.

- Navigate to your Terraform Cloud `devops-aws-myapp-dev` in the UI.
- Once there, navigate to the `Variables` tab.
- Validate that there is a Terraform variable named `environment` with a value of `development`

Repeat these same steps for both the `devops-aws-myapp-prod`

- Navigate to your Terraform Cloud `devops-aws-myapp-prod` in the UI.
- Once there, navigate to the `Variables` tab.
- Validate that there is a Terraform variable named `environment` with a value of `production`
- Task 5: Use the VCS workflow to deploy an update to development
- Task 6: Promote changes via GitOps to production

## Task 5: Update TFC development workspace to use development branch

While each individual on a team still makes changes to Terraform configuration in their editor of choice, they save their changes to version control branches to avoid colliding with each other's work. Working in branches enables team members to resolve mutually incompatible infrastructure changes using their normal merge conflict workflow.

### 5.1 Create a Development Branch

In the `my-app` github repository, create a `development` branch from the `main` branch. Update your Terraform Cloud `devops-aws-myapp-dev` workspace to point to your `development` branch.

### 5.2 Connect Terraform Cloud Workspace to development branch

1. Navigate to <https://app.terraform.io> and click the `devops-aws-myapp-dev` workspace
2. Select Settings » Version Control
3. Click the “Connect to version control”
4. Select the Version Control workflow
5. Click the VCS Connection in the “Source” section.
6. Verify you can see repositories and select the `my-app` github repository.





7. Specify `development` for the VCS Branch. The `VCS branch` will denote which branch in GitHub that this workspace will be watching for changes. Since this is the development workspace we will connect it to the `development` branch of our repository.
8. Select Update VCS Settings
9. Validate that a new Terraform run will occur on the workspace. Confirm and Apply the Terraform run.

## Task 6: Use the VCS workflow to deploy an update to development

### Version Control Branching and Terraform Cloud Workspaces

We will be using GitHub to promote a change to our app through Development and then into Production. Let's look how a change promotion would look in this configuration we outlined. We are going to start in our "Development" environment and move, or promote, that change to our production environments.

#### 6.1 Pull down development branch locally:

The first step is to pull down the development code locally from the development branch of our repository. Currently this branch matches the main branch.

```
git branch -f development origin/development  
git checkout development
```

Note: If you are uncomfortable with the `git` commands you can choose to make your edits in GitHub directly through the web editor. This will limit some of the validation and plan commands

#### 6.2 Perform infrastructure updates and refactor terraform code

Let's make a few changes in our development environment.

1. Removing additional Ubuntu server.
2. Remove the associate output blocks for this server.
3. Refactor our code to rename and move our application output blocks to an `outputs.tf` file

To remove the additional server, remove it's resource block from your `main.tf`





```
# Terraform Resource Block - To Build EC2 instance in Public Subnet
resource "aws_instance" "ubuntu_server" {
    ami                      = data.aws_ami.ubuntu.id
    instance_type            = "t2.micro"
    subnet_id                = aws_subnet.public_subnets["public_subnet_1"].id
    security_groups          = [aws_security_group.vpc-ping.id,
                                aws_security_group.ingress-ssh.id, aws_security_group.vpc-web.id]
    associate_public_ip_address = true
    key_name                 = aws_key_pair.generated.key_name
    connection {
        user      = "ubuntu"
        private_key = tls_private_key.generated.private_key_pem
        host       = self.public_ip
    }

    # Leave the first part of the block unchanged and create our `local-exec` provisioner
    # provisioner "local-exec" {
    #     command = "chmod 600 ${local_file.private_key_pem.filename}"
    # }

    provisioner "remote-exec" {
        inline = [
            "sudo rm -rf /tmp",
            "sudo git clone https://github.com/hashicorp/demo-terraform-101 /tmp",
            "sudo sh /tmp/assets/setup-web.sh",
        ]
    }

    tags = {
        Name = "Ubuntu EC2 Server"
    }

    lifecycle {
        ignore_changes = [security_groups]
    }
}
```

Let's also remove the `output` blocks that were referencing this resource block. Remove the following output blocks from your `main.tf`

```
output "public_ip" {
    value = aws_instance.ubuntu_server.public_ip
}

output "public_dns" {
```





```
    value = aws_instance.ubuntu_server.public_dns
}
```

The last change will be to refactor our code to rename our output blocks and move them to an `outputs.tf` file. Remove the following blocks from our `main.tf`:

```
output "public_ip_server_subnet_1" {
  value = aws_instance.web_server.public_ip
}

output "public_dns_server_subnet_1" {
  value = aws_instance.web_server.public_dns
}
```

Create an `outputs.tf` file within our working directory with the following output blocks.

```
output "environment" {
  value = var.environment
}

output "public_ip_web_app" {
  value = aws_instance.web_server.public_ip
}

output "public_dns_web_app" {
  value = aws_instance.web_server.public_dns
}
```

Be sure that all of our changes are valid by performing a `terraform init` and `terraform validate`

```
terraform init -backend-config=dev.hcl -reconfigure
terraform validate
```

```
Success! The configuration is valid.
```

Now we can run a `terraform plan` to see the impact of our changes. Take notice what happens when we try to issue an apply for a workspace that is VCS connected.

```
terraform plan
terraform apply
```

```
| Error: Apply not allowed for workspaces with a VCS connection
|
| A workspace that is connected to a VCS requires the VCS-driven workflow
| to ensure that
| the VCS remains the single source of truth.
```





This is to be expected. Now that we are centrally storing all of our Terraform configuration inside version control, and have connected our Terraform Cloud workspaces to the version control workflow, all updates should be triggered by committing our code up into GitHub.

### 6.3 Commit changes to development branch

Now that our changes are valid, let's commit them to our development branch. Before we do, make sure you have both GitHub and Terraform Cloud web pages up to see the change being committed to the development branch with then triggers a Terraform Run inside our `devop-aws-myapp-dev` workspace.

```
git add .
git commit -m "remove extra server and refactor outputs"
git push
```

The screenshot shows a GitHub repository page for 'gmaentz / my-app'. The repository is private. The main navigation bar includes 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the bar, there are links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Security', 'Insights', and 'Settings'. The 'Code' tab is selected. In the top right, there are buttons for 'Unwatch' (with 1 follower) and 'Fork'. The repository has 2 branches and 0 tags. A green 'Code' button is visible. The commit history shows one commit from 'gmaentz' titled 'remove extra server and refactor outputs' (490abe3, 1 hour ago). Below the commit, there are file commits for '.gitignore', 'README.md', 'main.tf', 'outputs.tf', and 'variables.tf'. On the right side, there are sections for 'About' (no description), 'Readme', 'Stars' (0), 'Watching' (1), 'Forks' (0), 'Releases' (no releases), and 'Packages' (no packages published).

**Figure 2:** Development Branch - Code Commit



# HashiCorp Certified: Terraform Associate

## Hands-On Labs



### 6.4 View Terraform Cloud VCS Workflow

Navigate to Terraform Cloud. You should see that a new run has been triggered within your `devops-aws-myapp-dev` workspace. You can view the details for the run to see that this was triggered by the code commit we just made on the development branch of our repository.

Workspaces 4 total				<a href="#">+ New workspace</a>
WORKSPACE NAME	RUN STATUS	REPO	LATEST CHANGE	
<a href="#">devops-aws-myapp-dev</a>	<span>⌚ Planning</span>	gmaentz/my-app	19 minutes ago	
<a href="#">devops-aws-myapp-prod</a>	<span>✓ Applied</span>	gmaentz/my-app	18 minutes ago	
<a href="#">getting-started</a>	<span>✓ Applied</span>		a month ago	
<a href="#">my-aws-app</a>	<span>✓ Applied</span>		2 days ago	

**Figure 3:** Development Workspace - Automatic Run

A screenshot of the Terraform Cloud interface showing a successful plan run. The top bar indicates a green checkmark for 'Plan finished' 3 minutes ago. Below it, the status bar shows 'Started 4 minutes ago > Finished 3 minutes ago'. A red bar at the bottom indicates '1 to destroy'. The main content area shows the AWS instance configuration with outputs like environment, public\_dns, and public\_ip. A note at the bottom right says 'Sentinel mocks can be used for testing your Sentinel policies'.

Plan finished 3 minutes ago

Started 4 minutes ago > Finished 3 minutes ago

– 1 to destroy

Filter resources by address... Resources: 0 to add, 0 to change, 1 to destroy ▾

Terraform 1.1.2 Download raw log

> — aws aws\_instance.ubuntu\_server

Outputs 7 planned to change

+	environment	"development"
-	public_dns	"ec2-44-201-20-195.compute-1.amazonaws.com"
-	public_dns_server_subnet_1	"ec2-3-237-201-55.compute-1.amazonaws.com"
+	public_dns_web_app	"ec2-3-237-201-55.compute-1.amazonaws.com"
-	public_ip	"44.201.20.195"
-	public_ip_server_subnet_1	"3.237.201.55"
+	public_ip_web_app	"3.237.201.55"

Download Sentinel mocks ⓘ Sentinel mocks can be used for testing your Sentinel policies

**Figure 4:** Development Workspace - Run Details

### 6.5 Confirm and apply our changes to development.

We can confirm and apply our changes to development environment from within Terraform Cloud.



# HashiCorp Certified: Terraform Associate

## Hands-On Labs

A screenshot of the HashiCorp Terraform Cloud interface. At the top, it shows a workspace named "devops-aws-myapp-dev" with 30 resources, Terraform version 1.1.2, and was updated 2 minutes ago. Below this is a navigation bar with tabs: Overview, Runs (which is selected), States, Variables, and Settings. To the right of the navigation is an "Actions" button. The main area displays a list of recent runs. The first run, triggered by "gmaentz" from GitHub 19 minutes ago, is shown in detail. It includes steps: "Plan finished" (4 minutes ago), "Cost estimation finished" (3 minutes ago), "Policy check passed" (3 minutes ago), and "Apply pending". A message at the bottom says "Needs Confirmation: Check the plan and confirm to apply it, or discard the run." with buttons for "Confirm & Apply", "Discard Run", and "Add Comment".

devops-aws-myapp-dev

No workspace description available. [Add workspace description](#).

Overview Runs States Variables Settings Actions

Resources 30 Terraform version 1.1.2 Updated 2 minutes ago

! Policy checked remove extra server and refactor outputs CURRENT

gmaentz triggered a run from GitHub 19 minutes ago Run Details

Plan finished 4 minutes ago Resources: 0 to add, 0 to change, 1 to destroy

Cost estimation finished 3 minutes ago Resources: 3 of 6 estimated - \$44.75/mo - -\$9.43

Policy check passed 3 minutes ago Policies: 2 passed, 0 failed

Apply pending

Needs Confirmation: Check the plan and confirm to apply it, or discard the run.

Confirm & Apply Discard Run Add Comment

**Figure 5:** Development Workspace - Confirm and Apply

### Task 7: Use the VCS workflow to merge an update to production

Now our changes have been applied in our development environment and we did our testing to confirm our application is functional, let's promote these changes to production. Let's look how a change promotion would look in this configuration we outlined.

1. Navigate back to GitHub. You should still be on the development branch with the ability to see the code changes that were made. We will be merging our changes we made in the development branch to our main branch. Click on the [Pull requests](#) option.
2. Select the [Compare & pull request](#) select [Change branches](#). Our source branch will default to “development”. Change the target branch to [main](#) and select [Create pull request](#)
3. Update the Title to [Promote to Production](#) and add a short description of your change.
4. For “Assignee” select [Assign to me](#). We currently do not have users and groups setup in our environment but in a real world scenario we can put security controls around this approval process.
5. On the bottom of the page you can view what files and lines in those files are different between the development and stage branches. These are the changes that will be merged into our target branch.





6. Select [Create pull request](#). We now have an opened a pull request. In our lab, approvals are optional but we could require multiple approvers before any changes get applied. We could deny the request and put a comment with details regarding why we denied it.
7. Click on the [Show all checks](#) next to the green check-mark.

The screenshot shows a GitHub repository interface for a branch named 'development'. The repository has 2 branches and 0 tags. The 'Code' tab is selected. A message indicates the branch is 1 commit behind master. The pull request list shows a single PR from 'gmaentz' with the commit message 'remove extra server and refactor outputs'. The PR has 2 commits and was pushed 19 minutes ago. The 'All checks have passed' section shows 2 successful checks: 'Terraform Cloud/exa...' and 'sentinel/example-or...'. The 'Details' link for each check is visible. The repository also contains files like .gitignore, README.md, main.tf, outputs.tf, and variables.tf. A file named 'README.md' is shown with an edit icon. The repository is titled 'My App'. On the right side, there's an 'About' section with no description, and sections for 'Readme', 'Stars', 'Watching', and 'Forks'. There are also sections for 'Releases' and 'Packages'.

**Figure 6:** GitHub/Terraform Cloud - Git Checks

8. Open the [Details](#) link in a new tab. As a pull request reviewer, you can use this to review the Terraform plan that was ran by the GitHub pipeline within the `devop-aws-myapp-dev` workspace.

We peer reviewed the changes everything looks good. Now go back to the tab we left open with our merge request and select the green [Merge pull request](#) button and [Confirm merge](#).

Notice that another pipeline was started under where the merge button was. Right click on this new pipeline and open it in a new tab. You can use the external pipeline to link out to Terraform Cloud to review the apply. We could have also been watching the Terraform Cloud workspace list to see our workspaces auto apply from our merge request inside the `devop-aws-myapp-prod` workspace.

You can validate and open the URL of the app to confirm that our changes have been added. Terraform Cloud VCS workflows allows us to promote our change from development into production environment





## Hands-On Labs

while maintaining isolation between these environments via Terraform workspaces. This ensures changes in the development branch have no impact to the production / main branch until a merge is performed.

### (Optional) Destroy infrastructure in appropriate workspace

If you would like to cleanup and destroy your infrastructure to keep costs down, that can be done at the workspace level. Navigate to the [Settings](#) of the workspace and select [Destruction and Deletion](#). That will provide you with the ability to [Queue destroy plan](#) which follows the same workflow as a [terraform destroy](#).





## Lab: Extending Terraform - Non Cloud Providers

In this challenge, you will use non cloud specific providers to assist in containing common tasks within Terraform.

- Task 1: Random Generator
- Task 2: SSH Public/Private Key Generator
- Task 3: Cleanup

### Create Folder Structure

Change directory into a folder specific to this challenge.

For example: /workstation/terraform/extending-terraform.

Create a `main.tf` file, we will add to this file as we go.

### Task 1: Random Generator

Create a random password:

```
resource "random_password" "password" {
  length  = 16
  special = true
}

output "password" {
  value = random_password.password.result
  sensitive = true
}
```

Run `terraform init`,

Run `terraform apply -auto-approve`.

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Outputs:

```
password = <sensitive>
```

Add the following configuration:





```
resource "random_uuid" "guid" {
}

output "guid" {
    value = random_uuid.guid.result
}
```

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

```
guid = 84502be0-13fc-4a49-ce02-4ab4e1b493ca
password = <sensitive>
```

Note: The password was NOT regenerated, why is this?

Now **update** the `random_guid` resource to use a “keepers” argument:

```
resource "random_uuid" "guid" {
    keepers = {
        datetime = timestamp()
    }
}
```

Run `terraform apply -auto-approve` several times in a row, what happens to the guid output?

## Task 2: SSH Public/Private Key Generator

Use Terraform to generate public/private SSH keys dynamically.

```
resource "tls_private_key" "tls" {
    algorithm = "RSA"
}
```

Run `terraform init`,

Run `terraform apply -auto-approve`.

Apply complete! Resources: 2 added, 0 changed, 2 destroyed.

Outputs:

```
guid = ad4efda0-d17d-559c-5cfe-1ed3ab9ce86b
password = <sensitive>
```





This is great, but I wan the keys as files, how?

Add the following config:

```
resource "local_file" "tls-public" {
  filename = "id_rsa.pub"
  content  = tls_private_key.tls.public_key_openssh
}

resource "local_file" "tls-private" {
  filename = "id_rsa.pem"
  content  = tls_private_key.tls.private_key_pem

  provisioner "local-exec" {
    command = "chmod 600 id_rsa.pem"
  }
}
```

What is this “local-exec”?

If you have too wide of permissions on a private SSH key, you can see the following error when trying to use that key to access a remote system:

```
Permissions 0777 for './id_rsa.pem' are too open.
It is recommended that your private key files are NOT accessible by others
.
This private key will be ignored.
```

This local exec will run a `chmod` on the file after it is created.

Run `terraform init`,

Run `terraform apply -auto-approve`.

You should now have two new files in your current working directory.

Now delete one of the files (i.e. `rm id_rsa.pem`).

Run a `terraform plan`, what changes (if any) are needed? Is this what you expected?

Run `terraform apply -auto-approve` to restore any deleted files.

### Task 3: Clean up

When you are done, run `terraform destroy` to remove everything (including the private and public key files).





## Lab: Terraform Auto Complete

One of the features of the Terraform CLI is that you can install a tab-completion if you are using `bash` or `zsh` as your command shell. This provides you with tab-completion support for all command names and *some* command arguments.

- Task 1: Install Terraform Auto Complete
- Task 2: Test Auto Complete via CLI
- Task 3: Install VSCode Extension for Auto Completion

### Task 1: Install Terraform Auto Complete

```
$ terraform -install-auto-complete
```

### Task 2: Test Auto Complete via CLI

In your terminal, type `terraform` and hit the `tab` key. Notice how auto complete has provided you a list of all the sub-commands available for use:

```
$ terraform
apply      destroy      fmt      get      import
login      output      providers   refresh   state
test       validate    workspace
console    env         force-unlock graph
logout     plan        push      show      init
untaint   version
```

Next, type `terraform state` and hit the `tab` key (note the space after `state`). Notice how auto complete has provided with additional options available for the `terraform state` command:

```
$ terraform state
list      mv      pull      push      replace-provider   rm      show
```

### Task 3: Install VSCode Extension for Auto Completion

In VSCode, select “Extensions” from the left navigation panel. Search for `terraform`. Install the `HashiCorp Terraform` extension verified from HashiCorp.





## Task 4: Use Auto Completion in VSCode

In your `main.tf` file, scroll to the bottom and add a few new lines. Type in `resource` (or just part of the word) and hit the tab key. See how the extension creates a new resource block for you and suggests resource types as you start typing. Type in `aws_ins` and you can tab complete on `aws_instance` to specify a new EC2 instance resource.

The screenshot shows a code editor window for a `main.tf` file. The code contains a partial `resource "aws_ins" "name" {` block. A dropdown auto-completion menu is open over the word `aws_ins`, listing various AWS resource types starting with `aws_`. The item `aws_instance` is highlighted with a blue selection bar. The menu also includes other resources like `aws_inspector_assessment_target`, `aws_inspector_assessment_template`, `aws_inspector_resource_group`, `aws_iam_instance_profile`, etc. At the bottom of the screen, the VSCode interface shows tabs for PROBLEMS (39), OUTPUT, and the current file `main.tf`.

```

164 | resource "aws_ins" "name" {
165 |   ...
166 |

```

- `aws_inspector_assessment_target`
- `aws_inspector_assessment_template`
- `aws_inspector_resource_group`
- `aws_instance` hashicorp/aws 3.64.2
- `aws_iam_instance_profile`
- `aws_iam_service_linked_role`
- `aws_iam_account_password_policy`
- `aws_iam_account_alias`
- `aws_iam_user_login_profile`
- `aws_imagebuilder_infrastructure_configuration`
- `aws_iam_user_policy_attachment`
- `aws_imagebuilder_distribution_configuration`

**Figure 1:** Auto Complete VSCode

## Troubleshooting

If you get an error message that says `Error executing CLI: Did not find any shells to install` you might need to update, or create, your `~/.zshrc` or `~/.profile` file to enable auto-complete. If you are using `zsh`, which is now the default on MacOS, you'll only need `~/.zshrc`. If you're using `bash`, you'll just need `~/.profile`. If this file does not yet exist, you can run the command:

```

# for zsh
touch ~/.zshrc

# for bash
touch ~/.profile

```

Try to install Terraform Auto Complete again.

If you get an error message that says `complete:13: command not found: compdef`, then you should add the following to the file mentioned above:





```
autoload -Uz compinit && compinit
```

With that now added, your `~/.zshrc` file should now look similar to this:

```
autoload -U +X bashcompinit && bashcompinit
autoload -Uz compinit && compinit
complete -o nospace -C /usr/local/bin/terraform terraform
```

You can restart your session or simply run `exec zsh` to reload the configuration.

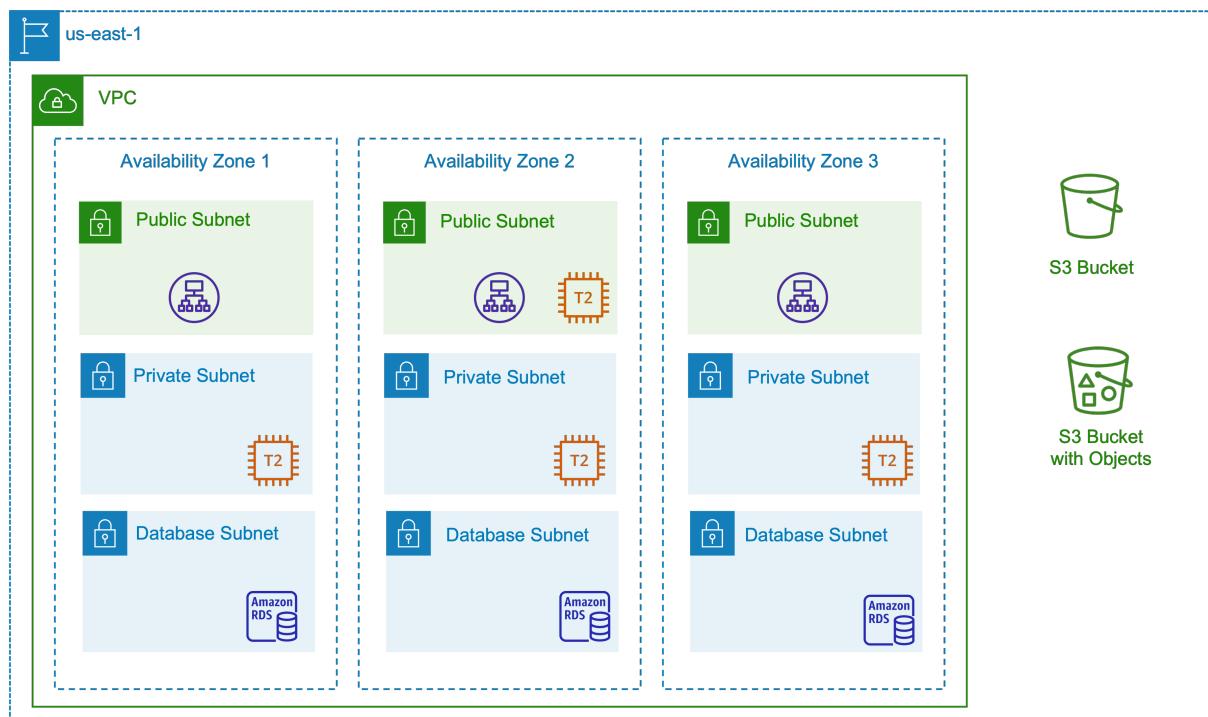




## Lab: Terraform AWS Provider - Multi-Region and Alias

The provider `alias` allows Terraform to differentiate the two AWS providers.

- Task 1: Configure Multiple AWS Providers
- Task 2: Build Resources using provider alias



**Figure 1:** AWS Application Infrastructure Buildout

### Task 1: Configure Multiple AWS Providers

```
provider "aws" {
  alias  = "east"
  region = "us-east-1"
}

provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```





## Task 2: Build Resources using provider alias

```
data "aws_ami" "ubuntu" {
  provider = aws.east
  most_recent = true

  filter {
    name      = "name"
    values    = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name      = "virtualization-type"
    values    = ["hvm"]
  }

  owners = ["099720109477"]
}

resource "aws_instance" "example" {
  provider = aws.east
  ami        = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  tags = {
    Name = "East Server"
  }
}

resource "aws_vpc" "west-vpc" {
  provider = aws.west
  cidr_block = "10.10.0.0/16"

  tags = {
    Name      = "west-vpc"
    Environment = "dr_environment"
    Terraform  = "true"
  }
}
```

## Reference

Use AssumeRole to Provision AWS Resources Across Accounts





## Exam Tips

If you're planning to take the HashiCorp Certified: Terraform Associate course, then this section is for you! We've put together some helpful tips that you can use to help prepare for the actual exam.

## About the Exam

The HashiCorp Certified: Terraform Associate exam includes multiple-choice, multi-select, and fill-in-the-blank type questions (not many of these). You'll be expected to take this proctored exam in a quiet space with a webcam enabled to ensure you are following the exam guidelines and not receiving additional assistance. The exam currently costs \$70.50 and is available to take through PSI. You can register for the exams by following the outline on [hashicorp.com/certifications](https://hashicorp.com/certifications).

## How Do I Prepare for the Exam?

While it might be obvious, the goal of this course is to go through all of the exam objectives and ensure you are comfortable with all the topics that might be covered. HashiCorp frequently updates its exams, so it is possible you will get questions on a topic not fully discussed in this course. If you find this to be true, please let us know so we can properly cover the topics and add them to the course. Beyond this course, we've gathered some tips and additional resources that you can use as your prepare to take the real exam.

## Hands-On Practice

Let's be honest, when it comes to Terraform, there's no better way to study and prepare than to use the product in your own environment. There is absolutely no substitution for hands-on experience with Terraform (which is why we created this course). Use the labs included in this course to ensure you have covered all the topics and make sure you are 100% comfortable with them. It's not enough just to do the tasks in the labs, but it is essential to understand "why" things happen the way they do when you run a `terraform plan` or `terraform apply`. Play around with the code provided in each lab and look up the commands, look up the attributes for resources. If cost is a concern, use the available providers that interact with your local system, like the `local`, `tls`, or `random` provider. Purposely break things and figure out how to fix them.





## Use Practice Questions

When you are getting comfortable with the topics presented in this exam, take a look at the practice questions that we've included in the course. These are designed to be similar to the real exam and test your knowledge on the information in each objective. It's not enough to just get them right, but know exactly WHY the correct answer is correct. Even better, know why the incorrect answers are clearly wrong. That's the real power of using practice exams to study.

When you're ready for more practice questions, check out Bryan's practice exam on Udemy. He wrote the very first practice exam for the Terraform Associate exam and it's been used by thousands of people to help prepare and pass the exam. With close to 10,000 students, it still has a 4.5 rating. If you're interested, check out [github.com/btkrausen/hashicorp](https://github.com/btkrausen/hashicorp) for a link and available coupons. You won't be disappointed.

## Use the HashiCorp official documentation

HashiCorp is one of the few software vendors that go to great lengths to provide you direct links where you can study for the test. If you head over to [hashicorp.com/certifications](https://hashicorp.com/certifications) and click on the Terraform Associate exam, you can find a few great links to use.

- The Study Guide provides a comprehensive list of links that you can use to get additional information.
- The Review Guide does a great job of breaking down each objective in the Terraform Associate exam and providing links and tutorials you can use to study that specific objective. ***Really good resource here.***
- You can also find some practice questions on this page as well.

The last part of the documentation suggestions are just around the official docs. If you're still feeling uneasy about some topics, go to the official documentation and start reading. They do a great job of breaking down features and how they can be used.

## Use the Q&A Section of the Course

Both of us are natural instructors, and we love helping people learn new things. Luckily for you, we also deploy these tools for a living, so we've got tons of experience working with many large enterprises. If you have questions, please use Udemy's Q&A feature to ask questions. Keep in mind that while we can't provide technical support here, we can help answer questions about Terraform features or course materials.

