[Terraform 1](#_Toc138776184)

[What is Terraform. 1](#_Toc111501289)

[What is Infrastructure as Code with Terraform? 1](#_Toc1524823880)

[How Does Terraform Work? 1](#_Toc526893892)

[How to Use Terraform. 2](#_Toc1686883213)

[Standardize your deployment workflow 3](#_Toc1778230335)

[Benefits of Terraform with Azure. 4](#_Toc1186157276)

[Provider 4](#_Toc2053738136)

[Provider in Terraform. 4](#_Toc778874060)

[What Providers Do? 4](#_Toc1098369837)

[Terraform providers for Azure infrastructure. 5](#_Toc1059977775)

[Azure Export for Terraform 6](#_Toc348798580)

[What is azure export for terraform? 6](#_Toc1073924202)

[Microsoft Azure Export for Terraform 6](#_Toc1564051920)

[How it Works 7](#_Toc1799556155)

[Config 7](#_Toc1077858335)

[Benefits 7](#_Toc1427186829)

[How to Export an Azure resource 8](#_Toc1093932637)

[Export your first resources using Azure Export for Terraform 8](#_Toc1234631498)

[Verify the Export results 9](#_Toc1052743460)

[Clean up resources 9](#_Toc951038799)

[Build Infrastructure for Windows 9](#_Toc1986133366)

[Install the Azure CLI tool using PowerShell 9](#_Toc481008247)

[Authenticate using the Azure CLI 10](#_Toc516804343)

[Create a Service Principal 11](#_Toc1516098605)

[Set your environment variables 12](#_Toc1828392117)

[Write configuration 12](#_Toc73733519)

[Terraform Block 13](#_Toc1756718963)

[Providers 13](#_Toc113370233)

[Resource 14](#_Toc723384267)

[Initialize your Terraform configuration 14](#_Toc85710564)

[Format and validate the configuration 14](#_Toc692417203)

[Apply your Terraform Configuration 15](#_Toc1252995602)

[Inspect your state 16](#_Toc1801451428)

[Change infrastructure 17](#_Toc397453240)

[Create a new resource 17](#_Toc598858488)

[Apply your changes 18](#_Toc61291666)

[Modify an existing resource 20](#_Toc1370230907)

[Apply your changes 20](#_Toc817507056)

[Review updates to state 22](#_Toc2105707247)

[Destroy infrastructure 22](#_Toc1607096697)

[Destroy 23](#_Toc966760731)

[ANNNEXURE 25](#_Toc1188373703)

[Terraform providers – General Introduction 25](#_Toc962434299)

[FAQ 26](#_Toc1018727104)

[Can I use multiple providers in one configuration file? 26](#_Toc675063998)

[Can I use multiple modules in one configuration file? 27](#_Toc1349009366)

[How Terraform Modules can accept input variables to customize their behavior. 29](#_Toc685831932)

[How terraform modules can produce outputs that can be used by other parts of your configuration. 31](#_Toc1271344636)

# Terraform

## What is Terraform.

Terraform is an open-source Infrastructure as Code (IaC) tool created by HashiCorp. It enables users to define and provision infrastructure resources such as virtual machines, networks, storage, and more, using a declarative configuration language.

Terraform configurations are written in HashiCorp Configuration Language (HCL) or optionally JSON, and they describe the desired state of the infrastructure.

## What is Infrastructure as Code with Terraform?

Infrastructure as Code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

Using Terraform has several advantages over manually managing your infrastructure:

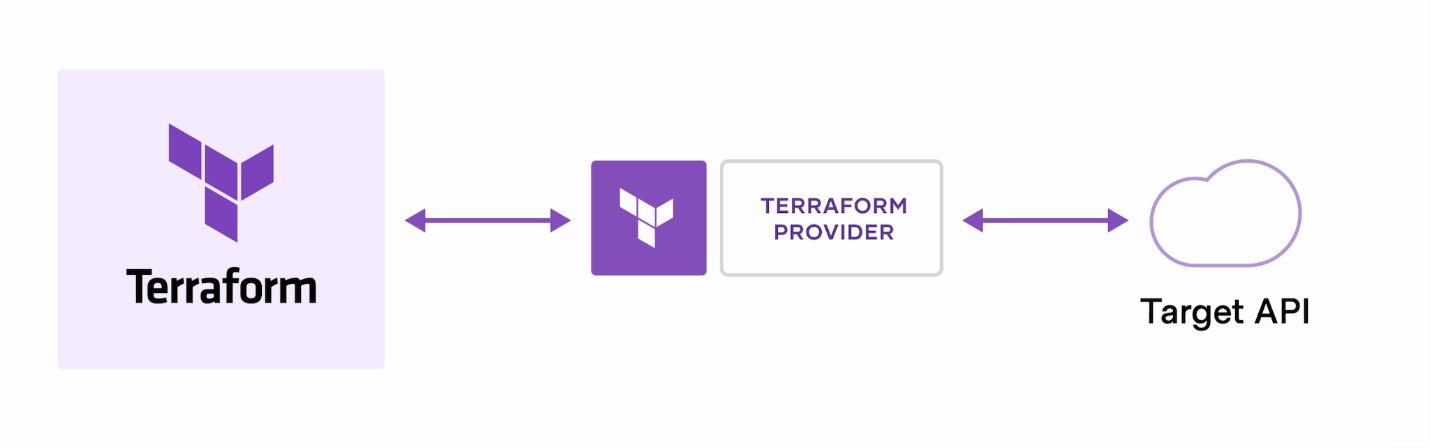
Terraform can manage infrastructure on multiple cloud platforms.

The human-readable configuration language helps you write infrastructure code quickly.

Terraform's state allows you to track resource changes throughout your deployments.

You can commit your configurations to version control to safely collaborate on infrastructure.

## How Does Terraform Work?

Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.

The core Terraform workflow consists of three stages:

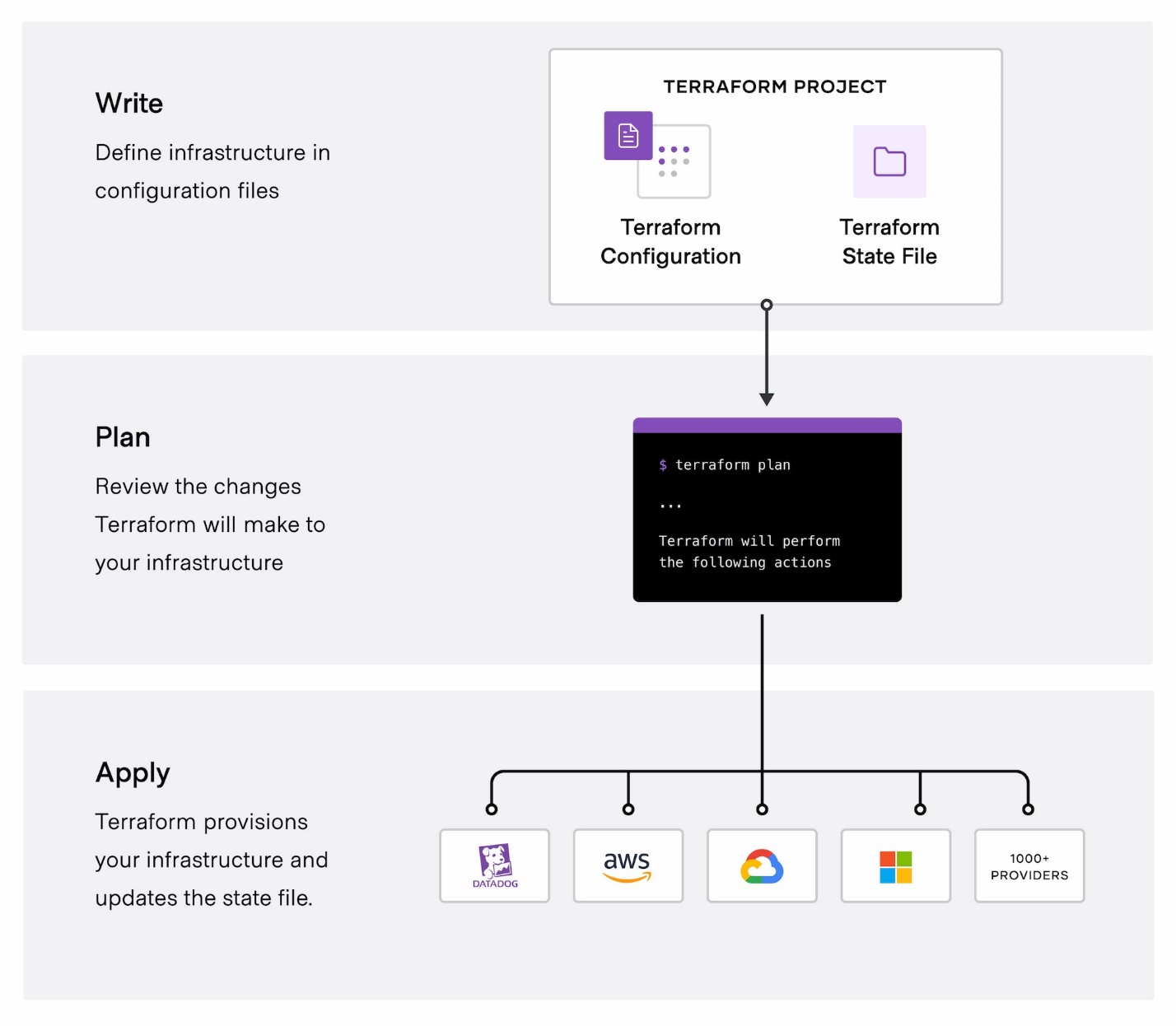
**Write:** You define resources, which may be across multiple cloud providers and services.

For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.

**Plan:** Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.

**Apply:** On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies.

For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.



## How to Use Terraform.

1. **Automated Infrastructure Provision**

Terraform enables you to automate the provisioning and management of infrastructure resources. This saves time and reduces the likelihood of human error associated with manual provisioning processes.

1. **Consistency and Version Control**

In the past, managing infrastructure configurations could lead to inconsistencies due to subtle differences introduced into the codebase.

Terraform helps maintain consistent configurations across different environments by defining infrastructure as code.

Version control ensures that changes are tracked and managed effectively.

1. **Infrastructure as Code (IaC)**

Terraform is part of the IaC movement, which aims to manage and provision infrastructure through code.

Tools like Terraform, Ansible, CloudFormation, Chef, and Puppet emerged to support IaC, shaping modern DevOps practices.

1. **Multi-Cloud and Hybrid Cloud Support**

Terraform supports multiple cloud providers, including AWS, Azure, Google Cloud Platform, and others, as well as on-premises infrastructure. This allows you to manage resources across different cloud providers or hybrid cloud environments using a single tool.

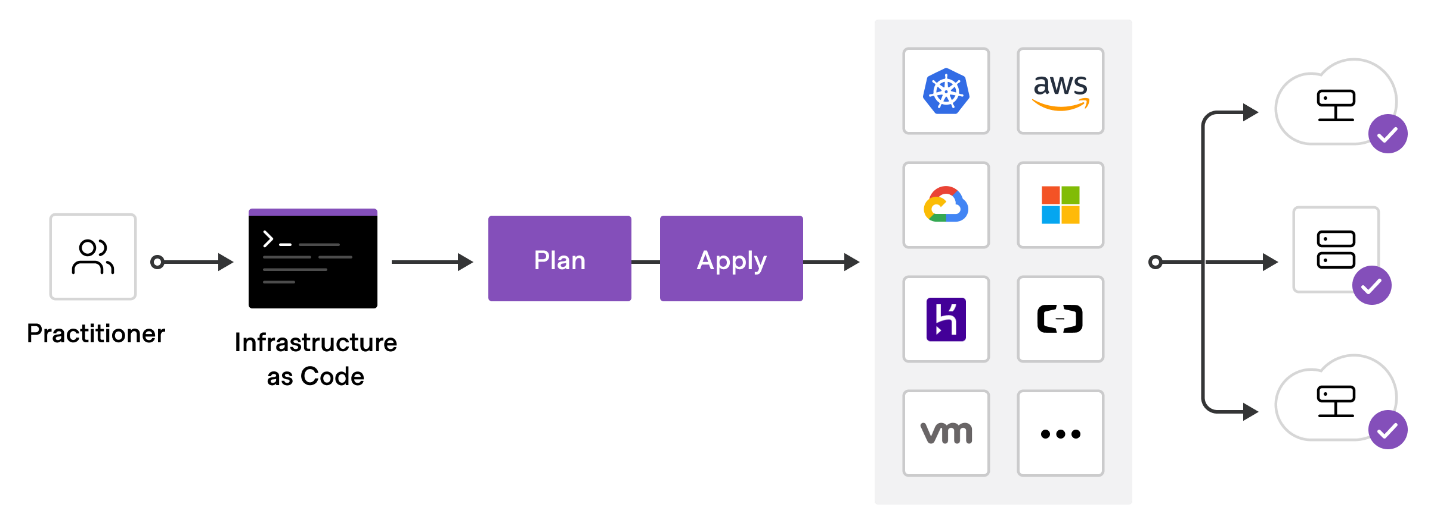
1. **Module Reusability**

Terraform offers modules, which are akin to functions or methods in programming languages.

Modules allow you to package and reuse common code, providing a standard interface for creating resources.

## Standardize your deployment workflow

Terraform's configuration language is declarative, meaning that it describes the desired end-state for your infrastructure, in contrast to procedural programming languages that require step-by-step instructions to perform tasks. Terraform providers automatically calculate dependencies between resources to create or destroy them in the correct order.

To deploy infrastructure with Terraform:

1. Scope - Identify the infrastructure for your project.
2. Author - Write the configuration for your infrastructure.
3. Initialize - Install the plugins Terraform needs to manage the infrastructure.
4. Plan - Preview the changes Terraform will make to match your configuration.
5. Apply - Make the planned changes.

## Benefits of Terraform with Azure.

1. **Common IaC tool**

Terraform Azure providers let you manage all your Azure infrastructure using the same declarative syntax and tooling. Using these providers, you can:

1. Configure core platform capabilities such as management groups, policies, users, groups, and policies.
2. Configure Azure DevOps projects and pipelines to automate regular infrastructure and application deployments.
3. Deploy Azure resources required by your applications.

**2. Scalability and Flexibility**

Terraform supports the dynamic creation and scaling of Azure resources based on your application's needs. You can easily scale resources up or down, adjust configurations, and add new resources as required, ensuring your infrastructure remains agile and responsive.

**3. Automate infrastructure management**

The Terraform template-based configuration file syntax enables you to configure Azure resources in a repeatable and predictable manner. Automating infrastructure includes the following benefits:

Lowers the potential for human errors while deploying and managing infrastructure.

Deploys the same template multiple times to create identical development, test, and production environments.

Reduces the cost of development and test environments by creating them on-demand.

**4. Integration with Azure Services**

Terraform provides first-class support for a wide range of Azure services, including virtual machines, storage accounts, databases, networking components, and more. This enables you to leverage the full capabilities of Azure while managing your infrastructure with Terraform.

# Provider

## Provider in Terraform.

Terraform relies on plugins called providers to interact with cloud providers, SaaS providers, and other APIs.

Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

What Providers Do?

Each provider adds a set of resource types and/or data sources that Terraform can manage.

Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.

Most providers configure a specific infrastructure platform (either cloud or self-hosted). Providers can

also offer local utilities for tasks like generating random numbers for unique resource names.

## Terraform providers for Azure infrastructure.

* **AzureRM:** Manage stable Azure resources and functionality such as virtual machines, storage accounts, and networking interfaces.
* **AzureAD:** Manage Microsoft Entra resources such as groups, users, service principals, and applications.
* **AzureDevops:** Manage Azure DevOps resources such as agents, repositories, projects, pipelines, and queries.
* **AzAPI:** Manage Azure resources and functionality using the Azure Resource Manager APIs directly. This provider compliments the AzureRM provider by enabling the management of Azure resources that aren't released.
* **AzureStack:** Manage Azure Stack Hub resources such as virtual machines, DNS, virtual networks, and storage.

1. **AzureRM Provider**:

The **AzureRM provider** (officially known as azurerm) is the primary Terraform provider for Azure.

It enables you to manage stable Azure resources and functionality, including virtual machines, storage accounts, networking interfaces, and more.

To use the AzureRM provider, configure it in your Terraform configuration like this:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "=3.0.0"

}

}

}

provider "azurerm" {

# Additional configuration options go here

}

You can find detailed documentation on the [Terraform Registry Azure Provider page](https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs).

1. **Authentication Methods**:

* Terraform supports various methods for authenticating to Azure:
  + **Azure CLI**: Use the Azure CLI for local development.
  + **Managed Service Identity (MSI)**: Ideal for non-interactive scenarios (e.g., CI/CD pipelines).
  + **Service Principal with Client Certificate or Client Secret**: Useful for automation.
  + **OpenID Connect**: For specific use cases.
* Ensure that the principal running Terraform has permissions to register Azure Resource Providers.

1. **Example Usage**:

Here’s a simple example of creating a resource group and a virtual network using the AzureRM provider:

provider "azurerm" {

skip\_provider\_registration = true

# Additional configuration options go here

}

resource "azurerm\_resource\_group" "example" {

name = "example-resources"

location = "West Europe"

}

resource "azurerm\_virtual\_network" "example" {

name = "example-network"

resource\_group\_name = azurerm\_resource\_group.example.name

location = azurerm\_resource\_group.example.location

address\_space = ["10.0.0.0/16"]

}

Adjust the configuration according to your needs.

# Azure Export for Terraform

## What is azure export for terraform?

Azure Export for Terraform is a powerful tool that simplifies the process of migrating Azure resources to Terraform. With just a single command, we can achieve the following:

Export Resources: Specify the set of Azure resources you want to export, and the tool generates Terraform HCL (HashiCorp Configuration Language) code and state files. This makes it easy to manage your infrastructure as code.

Inspect Infrastructure: You can also inspect existing Azure infrastructure, including all exposed properties. This helps you understand the current state of your resources before transitioning to Terraform.

Plan and Apply Workflow: Integrate non-Terraform infrastructure into your Terraform setup by following the familiar plan/apply workflow. This ensures a smooth transition and consistent management of your resources.

Microsoft Azure Export for Terraform

**Goal**

Azure Export for Terraform exports supported resources into Terraform state and generate the corresponding Terraform configuration. Both the Terraform state and configuration are expected to be consistent with the resources' remote state, i.e., terraform plan shows no diff. The user can use Terraform to manage these resources.

**Install**

**From Package Manager**

* **Windows**

   winget install aztfexport

* **Homebrew (Linux/macOS)**

   brew install aztfexport

## How it Works

aztfexport leverages aztft to identify the Terraform resource type corresponding to an Azure resource ID.

Then it runs terraform import under the hood to import each resource. Afterwards, it runs tfadd to generate the Terraform HCL code for each imported resource.

## Config

aztfexport will create a configuration file at $HOME/.aztfexport/config.json. This file is aim to be managed by command aztfexport config [subcommand], which includes following subcommands:

**get :** Get a config item

**set :** Set a config item

**show :** Show the full configuration

Currently, the following config items are supported:

* installation\_id : A UUID created on first run. If there is Azure CLI or Azure PowerShell installed on the current machine, the UUID will be the same value among these tools. Otherwise, a new one will be created. This is used as an identifier in the telemetry trace.
* telemetry\_enabled : Enables telemetry. We use telemetry to identify issues and areas for improvement to optimize this tool for better performance, reliability, and user experience. If you wish to disable our telemetry, set this to false.

## Benefits

Azure Export for Terraform enables you to:

* **Simplify migration to Terraform on Azure.** Azure Export for Terraform allows you to migrate Azure resources to Terraform using a single command.
* **Export user-specified sets of resources to Terraform HCL code and state with a single command.** Azure Export for Terraform enables you to specify a predetermined scope to export. The scope can be as granular as a single resource. You can also export a resource group and its nested resources. Finally, you can export an entire subscription.
* **Inspect preexisting infrastructure with all exposed properties.** Whether learning a newly released resource or investigating an issue in production, Azure Export for Terraform supports a read-only export with the option to expose all configurable resource properties.
* **Follow plan/apply workflow to integrate non-Terraform infrastructure into Terraform.** Export HCL code, inspect non-Terraform resources and easily integrate them into your production infrastructure and remote backends.

## How to Export an Azure resource

Using the interactive mode

1. Create a directory in which to test.
2. Open a command prompt and navigate to the new directory.
3. Run aztfexport resource-group to export the resource group named myResourceGroup. 
4. After the tool initializes, a list of the resources to be exported is displayed. Each line has an Azure resourceID matched to the corresponding AzureRM resource type.
5. Press w to run the export.

Note: For further information follow this link: [Quickstart: Export your first resources using Azure Export for Terraform | Microsoft Learn](https://learn.microsoft.com/en-us/azure/developer/terraform/azure-export-for-terraform/export-first-resources?tabs=azure-powershell)

## Export your first resources using Azure Export for Terraform

Create the test Azure resources

1. Run New-AzResourceGroup to create an Azure resource group.

A close-up of a sign

Description automatically generated

1. Run New-AzVM to create the virtual machine.

A screenshot of a computer program

Description automatically generated

## Verify the Export results

After the tool has finished exporting your Azure resources, run the following commands in the same directory that contains the generated files.

1. Run terraform init.



1. Run terraform plan.



If the terminal outputs **No changes needed**, then congratulations!

Your infrastructure and its corresponding state have been successfully exported to Terraform.

## Clean up resources

When you no longer need the resources:

1. Navigate to the directory containing your Terraform files for this article.
2. Run terraform destroy.

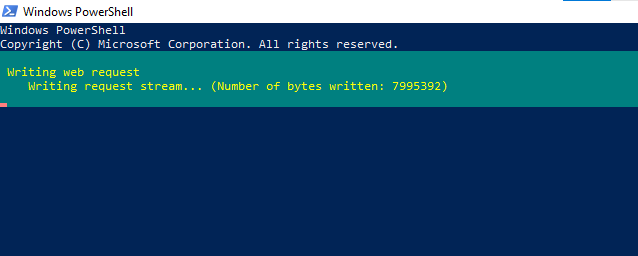
# Build Infrastructure for Windows

## Install the Azure CLI tool using PowerShell

Open your PowerShell prompt as an administrator and run the following command:

Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows -OutFile .\AzureCLI.msi; Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet'; rm .\AzureCLI.msi

After running the above code, you will get a window like this.



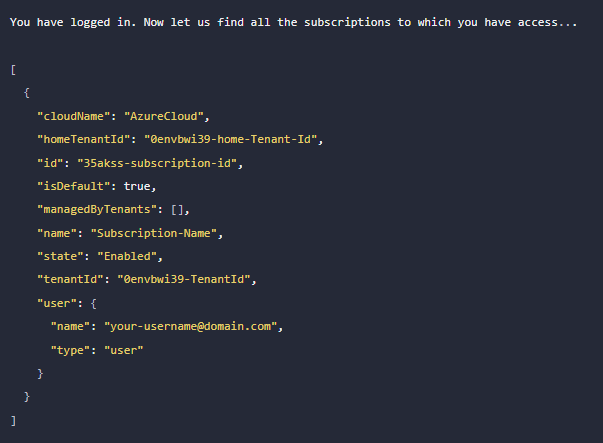
## Authenticate using the Azure CLI

Terraform must authenticate to Azure to create infrastructure.

In your terminal, use the Azure CLI tool to set up your account permissions locally.

az login

Your browser will open and prompt you to enter your Azure login credentials. After successful authentication, your terminal will display your subscription information.



Find the id column for the subscription account you want to use.

Once you have chosen the account subscription ID, set the account with the Azure CLI.

az account set --subscription "35akss-subscription-id"

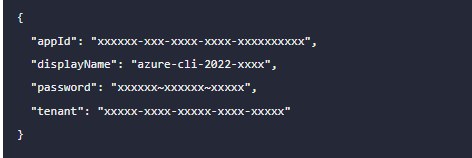
## Create a Service Principal

Next, create a Service Principal. A Service Principal is an application within Azure Active Directory with the authentication tokens Terraform needs to perform actions on your behalf. Update the <SUBSCRIPTION\_ID> with the subscription ID you specified in the previous step.

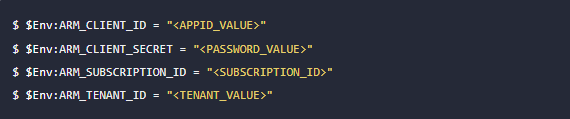
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/<SUBSCRIPTION\_ID>"

Creating 'Contributor' role assignment under scope '/subscriptions/35akss-subscription-id'

After running the above code you will get,



## Set your environment variables

In your PowerShell terminal, set the following environment variables. Be sure to update the variable values with the values Azure returned in the previous command.

## Write configuration

Create a folder called learn-terraform-azure.

New-Item -Path "c:\" -Name "learn-terraform-azure" -ItemType "directory"

Create a new file called main.tf and paste the configuration below.



Note

The location of your resource group is hardcoded in this example. If you do not have access to the resource group location westus2, update the main.tf file with your [Azure region](https://docs.microsoft.com/en-us/azure/virtual-machines/regions).

## Terraform Block

The terraform {} block contains Terraform settings, including the required providers Terraform will use to provision your infrastructure. For each provider, the source attribute defines an optional hostname, a namespace, and the provider type. Terraform installs providers from the [Terraform Registry](https://registry.terraform.io/) by default. In this example configuration, the azurerm provider's source is defined as hashicorp/azurerm, which is shorthand for registry.terraform.io/hashicorp/azurerm.

You can also define a version constraint for each provider in the required\_providers block. The version attribute is optional, but we recommend using it to enforce the provider version. Without it, Terraform will always use the latest version of the provider, which may introduce breaking changes.

## Providers

The provider block configures the specified provider, in this case azurerm. A provider is a plugin that Terraform uses to create and manage your resources. You can define multiple provider blocks in a Terraform configuration to manage resources from different providers.

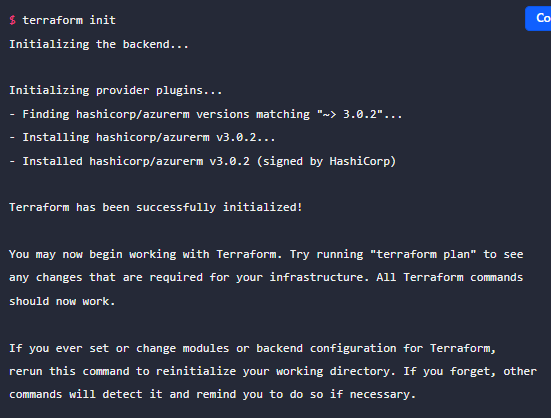
## Resource

Use resource blocks to define components of your infrastructure. A resource might be a physical component such as a server, or it can be a logical resource such as a Heroku application.

Resource blocks have two strings before the block: the resource type and the resource name. In this example, the resource type is azurerm\_resource\_group and the name is rg. The prefix of the type maps to the name of the provider. In the example configuration, Terraform manages the azurerm\_resource\_group resource with the azurerm provider. Together, the resource type and resource name form a unique ID for the resource. For example, the ID for your network is azurerm\_resource\_group.rg.

## Initialize your Terraform configuration

Initialize your learn-terraform-azure directory in your terminal. The terraform commands will work with any operating system. Your output should look like the one below.

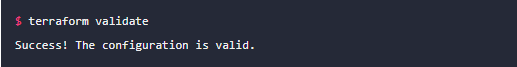


## Format and validate the configuration

The terraform fmt command automatically updates configurations in the current directory for readability and consistency.

Format your configuration. Terraform will print out the names of the files it modified, if any. In this case, your configuration file was already formatted correctly, so Terraform won't return any file names.



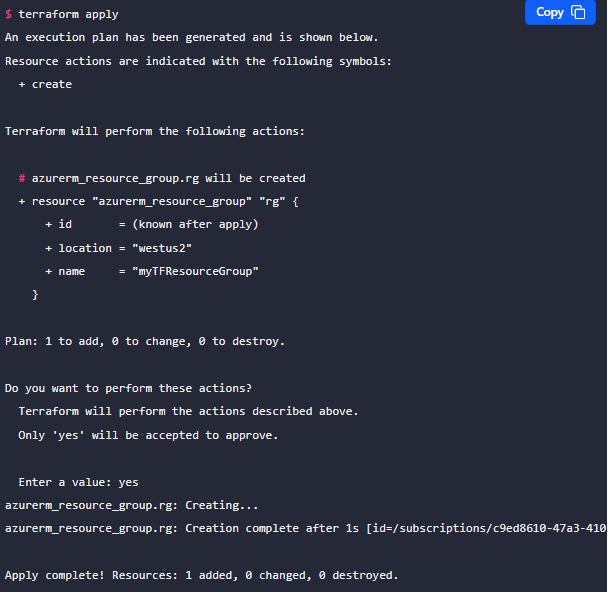


Validate your configuration. The example configuration provided above is valid, so Terraform will return a success message.

## Apply your Terraform Configuration

Run the terraform apply command to apply your configuration.

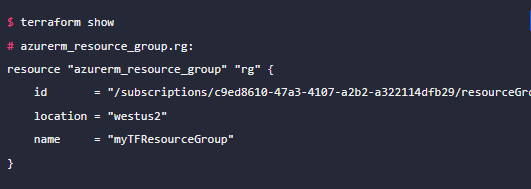
This output shows the execution plan and will prompt you for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure. Type yes at the confirmation prompt to proceed.



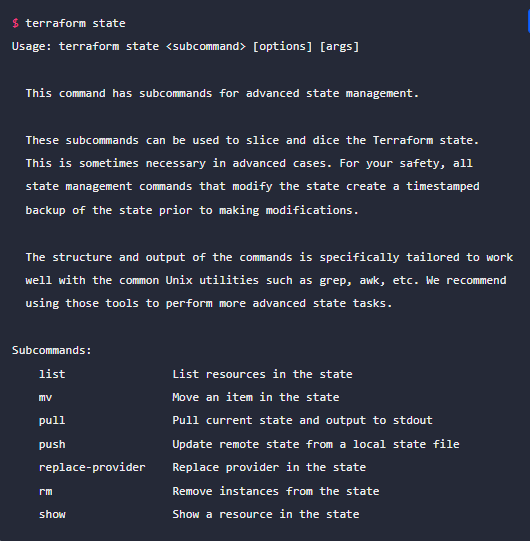
## Inspect your state

When you apply your configuration, Terraform writes data into a file called terraform.tfstate. This file contains the IDs and properties of the resources Terraform created so that it can manage or destroy those resources going forward.

Inspect the current state using terraform show.



If you run terraform state, you will see a full list of available commands to view and manipulate the configuration's state.



# Change infrastructure

## Create a new resource

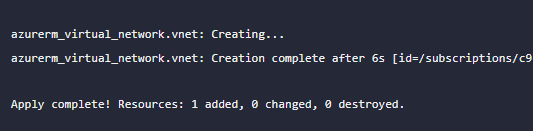
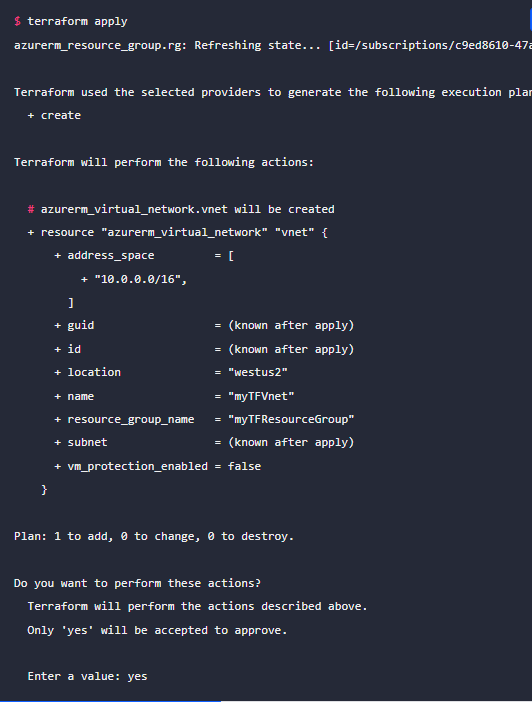
In your main.tf file, add the resource block below to create a virtual network (VNet).



To create a new Azure VNet, specify the name of the resource group to contain the VNet. By referencing the resource group, you establish a dependency between the resources. Terraform ensures that resources are created in proper order by constructing a dependency graph for your configuration.

## Apply your changes

After changing the configuration, run terraform apply again to see how Terraform will apply this change to your infrastructure. Respond yes to the prompt to confirm the changes.



Terraform builds an execution plan by comparing your desired state as described in the configuration to the current state, which is saved in either the local terraform.tfstate file or in a remote state backend depending on your configuration.

## Modify an existing resource

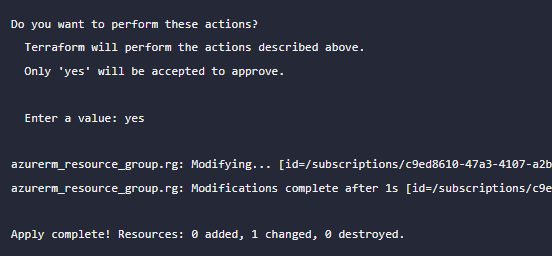
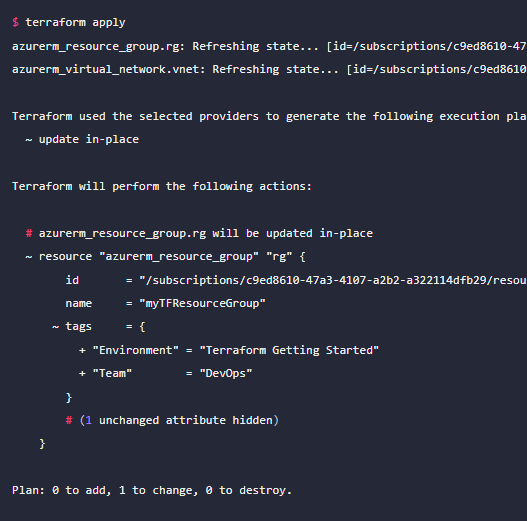
In addition to creating new resources, terraform can modify existing resources.

Open your main.tf file. Update the azurerm\_resource\_group resource in your configuration by adding the tags block as shown below:



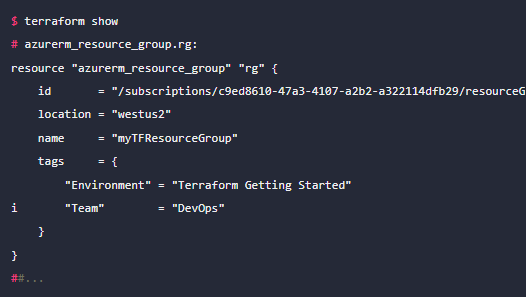
## Apply your changes

Run terraform apply to modify your infrastructure. Respond yes to the prompt to confirm the operation.

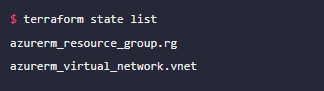


## Review updates to state

Use terraform show again to see the new values associated with this resource group.



Run terraform state list to get the updated list of resources managed in your workspace.

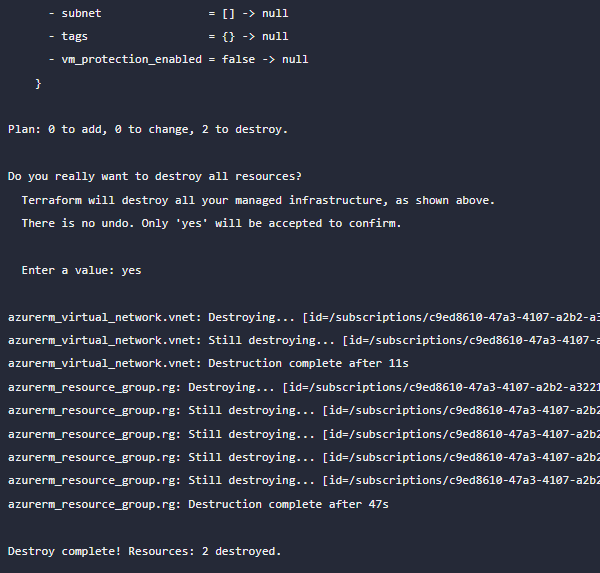
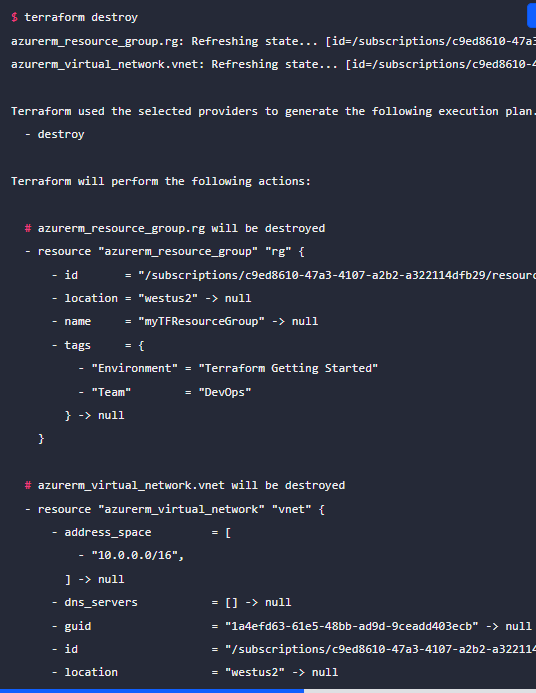


# Destroy infrastructure

Once you no longer need infrastructure, you may want to destroy it to reduce your security exposure and costs. For example, you may remove a production environment from service or manage short-lived environments like build or testing systems. In addition to building and modifying infrastructure, Terraform can destroy or recreate the infrastructure it manages.

## Destroy

The terraform destroy command terminates resources managed by your Terraform project. This command is the inverse of terraform apply in that it terminates all the resources specified in your Terraform state. It does *not* destroy resources running elsewhere that are not managed by the current Terraform project.

Destroy the resources you created. When terraform prompts you, type yes to execute this plan and destroy the infrastructure.

# ANNNEXURE

## Terraform providers – General Introduction

**Terraform providers**, which play a crucial role in managing infrastructure resources. Here’s what you need to know:

1. **What Are Terraform Providers?**

* A **provider** in Terraform is a **plugin** that enables interaction with an API.
* Providers serve as the bridge between Terraform and external services, cloud platforms, and software-as-a-service (SaaS) offerings.
* They allow you to define and manage resources (such as virtual machines, databases, networks, etc.) in your Terraform configuration.

1. **Types of Providers**

* **Cloud Providers**: These are the most common. Examples include **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)**.
* **Software-as-a-Service (SaaS) Providers**: These handle services like **GitHub**, **Splunk**, **DataDog**, and more.
* **Custom Providers**: You can even create your own custom providers for in-house tools or specialized services.

1. **Using Providers in Terraform Configuration**:

* In your Terraform configuration files (usually with a .tf extension), you specify the providers you want to use.
* Example:provider "aws" {  
   region = "us-west-2"  
  }
* This block tells Terraform to interact with AWS services in the specified region.

1. **Finding Providers**:

* The **Terraform Registry** hosts a wide range of publicly available providers.
* You can find providers for AWS, Azure, GCP, Kubernetes, Helm, and many others.
* Explore the [Terraform Registry Providers to discover and use the ones you need1](https://registry.terraform.io/browse/providers).

1. **Version Constraints**:

* By default, Terraform downloads the latest version of each provider.
* You can specify version constraints to ensure compatibility with your existing configurations.

1. **Provider Development** (Advanced):

* If you’re interested in creating your own provider, refer to the official Terraform Plugin Development documentation for detailed guidance.

# FAQ

## Can I use multiple providers in one configuration file?

You can **absolutely use multiple providers in a single Terraform configuration file**. This flexibility allows you to manage resources across different cloud platforms or services within the same infrastructure definition.

Here’s how you can do it:

1. **Provider Blocks**:

In your Terraform configuration (usually written in .tf files), define multiple provider blocks.

Each provider block specifies the type of provider (e.g., AWS, Azure, Google Cloud) and its configuration.

Example:

provider "aws" {

region = "us-west-2"

}

provider "azurerm" {

features {}

}

1. **Resource Configuration**:

Once you’ve defined your providers, you can create resources associated with each provider.

For instance, you can create an AWS EC2 instance using the AWS provider and an Azure virtual machine using the AzureRM provider.

Example:

resource "aws\_instance" "example" {

# AWS-specific configuration

# ...

}

resource "azurerm\_virtual\_machine" "example" {

# Azure-specific configuration

# ...

}

1. **Provider Aliases** (Optional):

If you have multiple providers of the same type (e.g., multiple AWS accounts), you can use provider aliases to differentiate them.

Example:

provider "aws" {

alias = "account1"

region = "us-west-2"

}

provider "aws" {

alias = "account2"

region = "us-east-1"

}

resource "aws\_instance" "example" {

provider = aws.account1

# ...

}

1. **Variables and Conditionals**:

You can use variables and conditionals to dynamically select providers based on input or environment.

For instance, you might choose different providers based on a variable value or a specific condition.

Example:

provider "aws" {

region = var.aws\_region

}

provider "azurerm" {

features {}

}

Remember to keep your configuration organized and maintain clear separation between different providers.

## Can I use multiple modules in one configuration file?

You can **definitely use multiple modules in a single Terraform configuration file**. In fact, modularization is a best practice for managing complex infrastructure definitions. Let’s dive into how you can achieve this:

1. **What Are Modules?**:
   * Modules in Terraform are reusable components that encapsulate a set of resources and their configurations.
   * They allow you to organize your infrastructure code into smaller, self-contained units.
   * Think of modules as building blocks that you can assemble to create your overall infrastructure.
2. **Creating and Using Modules**:

To create a module, create a new directory (usually named after the module) containing .tf files.

Inside the module directory, define the resources you want to encapsulate.

Example directory structure:

my\_project/

├── main.tf

├── variables.tf

├── outputs.tf

├── modules/

│ ├── vpc/

│ │ ├── main.tf

│ │ ├── variables.tf

│ │ └── outputs.tf

│ ├── app/

│ │ ├── main.tf

│ │ ├── variables.tf

│ │ └── outputs.tf

In your main.tf, use the module like this:

module "my\_vpc" {

source = "./modules/vpc"

# Additional input variables go here

}

module "my\_app" {

source = "./modules/app"

# Additional input variables go here

}

1. **Input Variables and Outputs**:

Modules can accept input variables (defined in variables.tf) to customize their behavior.

They also produce outputs (defined in outputs.tf) that can be used by other parts of your configuration.

Example:

# modules/vpc/variables.tf

variable "vpc\_cidr" {

description = "CIDR block for VPC"

type = string

}

# modules/vpc/main.tf

resource "aws\_vpc" "my\_vpc" {

cidr\_block = var.vpc\_cidr

# ...

}

# modules/vpc/outputs.tf

output "vpc\_id" {

value = aws\_vpc.my\_vpc.id

}

1. **Reusing Modules**:

* You can reuse the same module multiple times with different input values.
* For example, create multiple VPCs or application clusters using the same module.
* Customize each instance of the module by setting different input variables.

1. **Module Sources**:

Modules can be sourced from various locations:

* **Local Paths**: As shown in the example above (source = "./modules/vpc").
* **Git Repositories**: Use a Git URL as the source.
* **Terraform Registry**: Publicly available modules hosted on the Terraform Registry.

1. **Module Versioning**:

Specify the version of a module using the version argument.

Example:

module "my\_vpc" {

source = "git::https://github.com/myorg/my-vpc-module.git?ref=v1.2.0"

# ...

}

Remember, modularization simplifies maintenance, promotes reusability, and enhances collaboration.

## How Terraform Modules can accept input variables to customize their behavior.

**Terraform modules** can accept input variables to customize their behavior. These input variables allow you to parameterize your modules and make them more flexible. Here’s how it works:

1. **Defining Input Variables**:

In your module directory (where you define your module’s resources), create a file named variables.tf.

Inside this file, declare the input variables you want to use.

Example:

# modules/my\_module/variables.tf

variable "region" {

description = "The AWS region where resources will be created."

type = string

default = "us-east-1"

}

variable "instance\_type" {

description = "The EC2 instance type."

type = string

default = "t2.micro"

}

1. **Using Input Variables**:

In your module’s main configuration file (usually named main.tf), reference these input variables.

Example:

# modules/my\_module/main.tf

resource "aws\_instance" "my\_instance" {

ami = "ami-12345678"

instance\_type = var.instance\_type

availability\_zone = "${var.region}a"

# ...

}

1. **Passing Values to Modules**:

When you use the module in your main configuration (e.g., main.tf), provide values for the input variables.

Example:

# main.tf

module "my\_ec2\_instance" {

source = "./modules/my\_module"

region = "us-west-2"

instance\_type = "m5.large"

}

1. **Default Values**:

Input variables can have default values (as shown in the example above).

If you don’t explicitly provide a value when using the module, Terraform uses the default value.

1. **Variable Types**:

Input variables can have different types, such as string, number, list, map, etc.

Specify the appropriate type based on the expected input.

1. **Sensitive Variables**:

If an input variable contains sensitive information (like passwords or API keys), mark it as sensitive.

Example:

variable "db\_password" {

description = "Database password"

type = string

sensitive = true

}

Remember, input variables allow you to customize your modules and create reusable, parameterized infrastructure code.

## How terraform modules can produce outputs that can be used by other parts of your configuration.

In Terraform, **output values** play a crucial role in making information about your infrastructure available for other parts of your configuration. Let’s dive into how this works:

1. **Declaring Output Values**:

Each output value exported by a module must be declared using an output block. For example:

output "instance\_ip\_addr" {

value = aws\_instance.server.private\_ip

}

Here, instance\_ip\_addr is the name of the output, and its value is derived from the private\_ip attribute of an aws\_instance resource defined elsewhere in the module.

1. **Use Cases for Output Values**:

**Child Modules**: A child module can use outputs to expose a subset of its resource attributes to a parent module.

**Root Modules**: In a root module, outputs can be used to print certain values in the CLI output after running terraform apply.

**Remote State**: When using remote state, root module outputs can be accessed by other configurations via a terraform\_remote\_state data source.

**Resource Instances**: Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration.

1. **Accessing Child Module Outputs**:

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>. For instance, if a child module named web\_server declared an output named instance\_ip\_addr, you could access that value as module.web\_server.instance\_ip\_addr.

1. **Custom Condition Checks**:

You can use precondition blocks to specify guarantees about output data. For example, ensuring that an EC2 instance has an encrypted root volume before using its IP address as an output value.

[Remember that outputs are only rendered when Terraform applies your plan, not during terraform plan execution1](https://developer.hashicorp.com/terraform/language/values/outputs)[2](https://developer.hashicorp.com/terraform/tutorials/configuration-language/outputs). Outputs act as the interface between different modules, facilitating the passing of information across your infrastructure.