TERRAFORM

Terraform Resources Overview – Examples & Best Practices

Jack Roper

06 Nov 2023 · 27 min read

In this post, we will give an overview of ‘resources’ in Terraform, showing the syntax, types, arguments you can use, meta-arguments, and step-by-step guidance on creating a resource, along with some examples and best practices. Let’s get started!

We will cover:

What are Terraform resources?

Terraform resources are components within a Terraform configuration that represent infrastructure objects or services that need to be managed.

Resources in Terraform configurations define the desired state of various infrastructure elements, e.g. virtual machines, SQL databases, network security groups, etc.

Terraform resource syntax

The syntax for defining a resource in Terraform typically follows this pattern:

```
resource "resource_type" "resource_name" {
  # Configuration settings for the resource
  attribute1 = value1
  attribute2 = value2
  # ...
}
```

2. `"resource_type"` : This is the type of resource you want to create. For example, if you're creating an AWS EC2 instance with Terraform, the resource type would be `"aws_instance"` .
3. `"resource_name"` : This is a user-defined name for the resource block. It must be unique within your Terraform configuration. It's used as a reference to the resource elsewhere in your configuration.
4. `{}` : The opening and closing curly braces enclose the configuration settings for the resource. Inside the block, you define the attributes and their values for the resource.
5. `attribute1 = value1` , `attribute2 = value2` : These lines define the attributes of the resource and their corresponding values. Attributes are specific properties or settings for the resource. The values can be literals, references to other resources or variables, expressions, or function calls. You can look up the available attributes for your chosen resource on the Terraform resource documentation pages.

For example, an Azure resource group might look like the following:

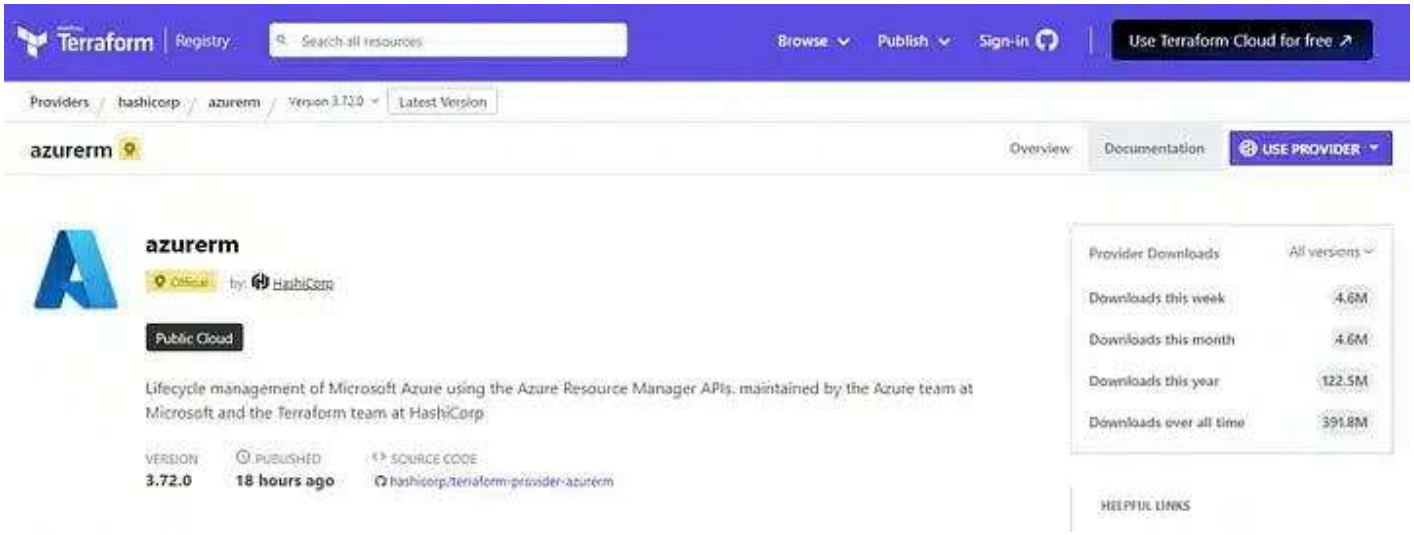
```
resource "azurerm_resource_group" "jacks-rg" {
  name     = "jacks-rg"
  location = "UK South"
}
```

How to use Terraform resources documentation

Using Terraform's resource documentation is essential when working with Terraform to understand the available resource types, their attributes, and how to configure them.

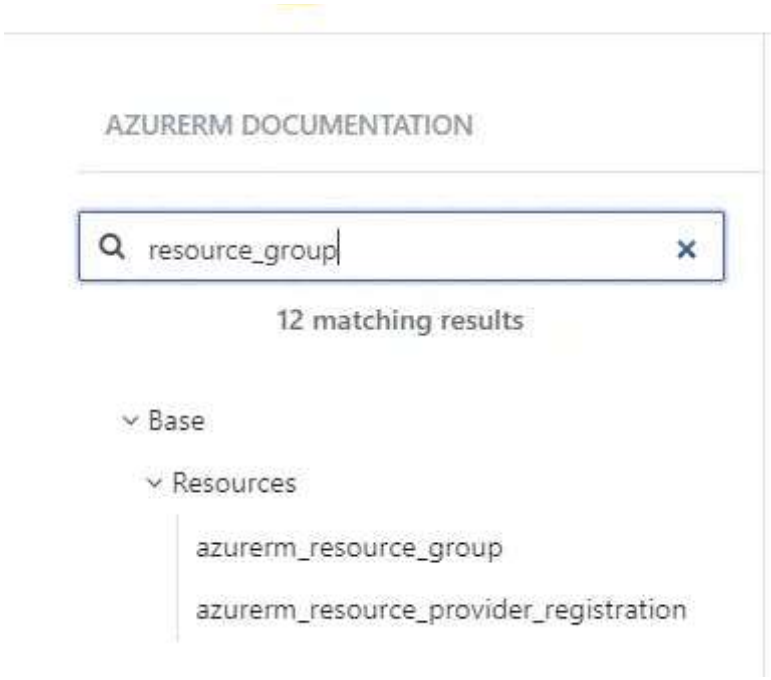
The official documentation is organized by providers, and each provider contains information about their resources.

Select the provider you are interested in, for example — Azure, and choose the documentation link at the top right.



2. Browse the resource list

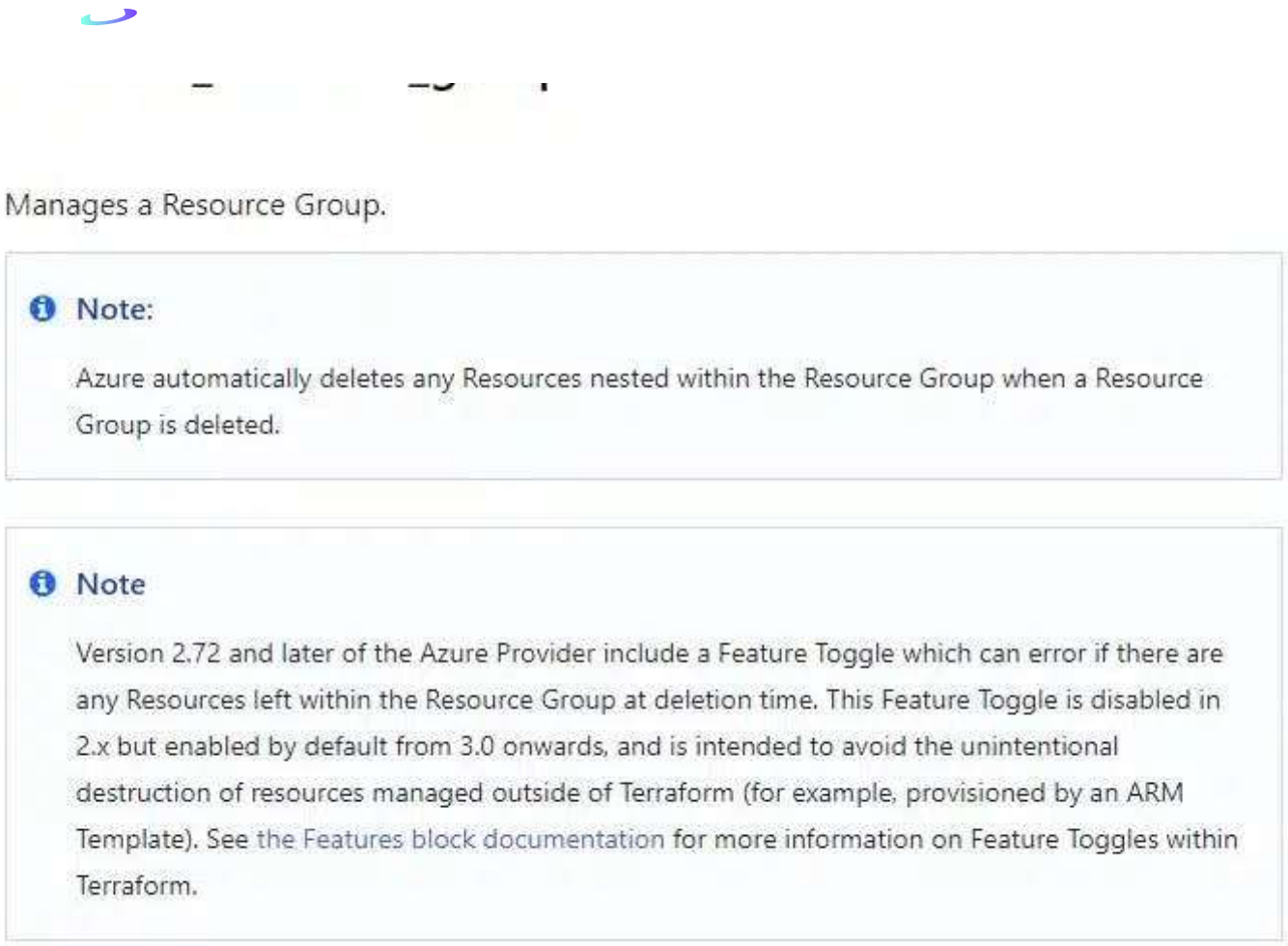
You will be presented with a list of resources you can use with Terraform, and you can search for the one you are interested in.



3. Select a resource

On selecting a resource, the resource page will show the available arguments you can use with the resource, along with argument references (values that are exported after you create the resource that you can reference from other parts of your code).

Most pages also include an example configuration for the resource, a timeouts section, and an example of how to import an already existing resource of that type into your Terraform state. Additional information will be presented here, too, such as notes of interest and warnings around the depreciation of particular arguments in certain Terraform versions.



Terraform resource types and arguments

As mentioned previously, Terraform resource types and their corresponding arguments vary depending on the provider being used, as Terraform supports a wide range of cloud and infrastructure providers.

A few examples are shown below, with a truncated list of available arguments for each.

Resource Type: `aws_s3_bucket` (Amazon S3 Bucket)

- `bucket` (string): The name of the S3 bucket.
- `acl` (string): The access control list (ACL) for the bucket.
- `versioning` (map): Configuration for bucket versioning.

- `name` (string): The name of the storage account.
- `resource_group_name` (string): The name of the resource group in which to create the storage account.
- `account_tier` (string): The performance tier of the storage account (e.g., “Standard” or “Premium”).
- `account_replication_type` (string): The replication type for the storage account (e.g., “LRS” or “GRS”).

Resource Type: `google_storage_bucket` (Google Cloud Storage Bucket)

- `name` (string): The name of the storage bucket.
- `location` (string): The location (region) of the bucket.
- `storage_class` (string): The storage class of the bucket (e.g., “STANDARD” or “COLDLINE”).

Terraform resources behavior

Terraform resources have specific behaviors and characteristics that define how they work within a Terraform configuration. Awareness of the key features of Terraform will help you to understand how Terraform resources behave.

Some key features include:

- Terraform resources are designed to be **idempotent**, meaning that applying the same configuration multiple times should not have unintended side effects.
- Resources can **depend** on each other. Terraform automatically determines the order in which resources should be created or modified based on these dependencies.
- Terraform uses a **declarative syntax**, which means you specify what you want the infrastructure to look like, not how to achieve that state.

configuration and apply it, Terraform will destroy that resource (if it exists) to match the new desired state.

- Terraform **maintains a state file** that tracks the current state of your infrastructure.
- Terraform is designed to be highly **parallelized**. It can create, modify, or destroy multiple resources simultaneously when possible, speeding up the provisioning process.
- Terraform can **detect and report any differences** between the current state of your infrastructure and the desired state defined in your configuration.
- Terraform supports **locking** to prevent concurrent modifications to the same infrastructure.

Accessing resources attributes

To access resource attributes from other places in your code, you can reference them directly.

For example, the Azure `resource_group` resource shows the following arguments and attributes references on the docs page:

The following arguments are supported:

- `location` - (Required) The Azure Region where the Resource Group should exist. Changing this forces a new Resource Group to be created.
- `name` - (Required) The Name which should be used for this Resource Group. Changing this forces a new Resource Group to be created.

- `managed_by` - (Optional) The ID of the resource or application that manages this Resource Group.
- `tags` - (Optional) A mapping of tags which should be assigned to the Resource Group.

Attributes Reference

In addition to the Arguments listed above - the following Attributes are exported:

- `id` - The ID of the Resource Group.

Our resource group configuration looks like the following:

```
resource "azurerm_resource_group" "jacks-rg" {
  name     = "jacks-rg"
```

We can reference the attributes directly in other resources, for example, an Azure virtual network resource:

```
resource "azurerm_virtual_network" "jacks-vnet" {
  name                = "jacks-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.jacks-rg.location
  resource_group_name = azurerm_resource_group.jacks-rg.name
}
```

You can also create output blocks in your Terraform configuration to expose specific attributes of resources for easy access. Output blocks define what attributes to expose and give them friendly names.

For example, to show the address space of the VNET:

```
output "address_space" {
  value = azurerm_virtual_network.jacks-vnet.address_space
}
```

You can then use `terraform output` to retrieve this value.

💡 You might also like:

- How to Improve Your Infrastructure as Code using Terraform

Resource dependencies

Resource dependencies refer to the relationships between different resources within your configuration. By default, Terraform automatically determines resource dependencies based on references in your configuration. When one resource references attributes of another resource, Terraform creates an implicit dependency.

In some cases, you may need to specify explicit dependencies using the Terraform `depends_on` argument within a resource block. This is useful when there are no direct attribute references but still a logical order of creation or modification.

For example, you could explicitly add the `depends_on` argument to our `azurerm_virtual_network` resource to instruct Terraform to make sure the resource group exists first — in this case, this is not necessary, as Terraform will create an implicit dependency anyway. As a best practice, you should avoid creating any unnecessary explicit dependencies and allow Terraform to manage them wherever possible.

```
resource "azurerm_virtual_network" "jacks-vnet" {
  name                = "jacks-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.jacks-rg.location
  resource_group_name = azurerm_resource_group.jacks-rg.name
  depends_on          = [azurerm_resource_group.jacks-rg]
}
```

Meta-Arguments

Meta-arguments are special configuration settings that can be applied to resource blocks, data blocks, and modules. Let's take a look at each one in turn with an example.

1. Depends_on

- **Syntax:** `depends_on = [resource1, resource2, ...]`
- **Usage:** Specifies explicit dependencies between resources. Terraform will ensure that the listed resources are created or modified before the current resource is processed.
- **Example:** See the example above.

2. Count

- **Syntax:** `count = n`
- **Usage:** Allows you to create multiple instances of a resource or module based on the specified count. This is useful when you want to create multiple similar resources, such as multiple EC2 instances or database replicas.
- **Example:**

```
resource "aws_instance" "example" {
  count = 3
  # Other configuration settings...
}
```

3. For_each

- **Syntax:** `for_each = { key1 = value1, key2 = value2, ... }`

pair represents a unique instance. This is useful when you want to create resources with distinct attributes or names. (Read more about Terraform for_each.)

- **Example:**

```
resource "aws_instance" "example" {
  for_each = {
    web1 = "t2.micro"
    web2 = "t2.micro"
    db   = "db.m3.medium"
  }
  instance_type = each.value
  # Other configuration settings...
}
```

4. Provider

- **Syntax:** `provider = aws`
- **Usage:** Allows you to specify which provider configuration should be used for a particular resource. This is helpful when you have multiple provider configurations defined in your Terraform configuration.
- **Example:**

```
resource "aws_instance" "example" {
  provider = aws.us-west-1
  # Other configuration settings...
}
```

5. Lifecycle

attributes. You can ignore changes, prevent replacements, and more using the `lifecycle` block.

- **Example:**

```
resource "aws_instance" "example" {
  # Configuration settings...

  lifecycle {
    ignore_changes = [tags]  # Ignore changes to tags attribute
  }
}
```

6. Provisioners

Terraform provisioners are used to execute scripts or commands on a remote resource (such as a virtual machine or cloud instance) after it has been created or updated. Provisioners help you perform tasks like configuring software, initializing databases, setting up networking, or any other custom operations needed to prepare a resource for use. There are two types of provisioners you can use:

- `local-exec` — Run scripts or commands on the machine where you're running Terraform. Typically used to initialize local services.

```
resource "aws_instance" "example" {
  ami           = "ami-0123456789abcdef0"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo 'Resource provisioned!'"
```

- `remote-exec` — Run scripts or commands on a remote resource over SSH. Typically used to configure and customize resources like virtual machines, instances, or containers. The example below shows how to use it to run a Powershell command on the provisioned Windows AWS instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0123456789abcdef0"
  instance_type = "t2.micro"

  connection {
    type        = "ssh"
    user        = "Administrator"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "powershell.exe -ExecutionPolicy Bypass -Command",
      "Write-Host 'Running PowerShell remotely'",
      "Get-Process | Select-Object -First 5  # Example PowerShell command"
    ]
  }
}
```

Local-only resources in Terraform

Local-only resources are defined using the `null_resource` resource type. These are resources that are managed solely within the Terraform state and do not correspond to any

The below example shows how to create a `null_resource` that includes a “always_run” trigger, which is set to the current timestamp using the `${timestamp()}` function. This trigger ensures that the local resource is recreated on each `terraform apply`, even if nothing else in the configuration has changed.

```
resource "null_resource" "example" {
  triggers = {
    # This trigger causes the resource to be recreated on each apply
    always_run = "${timestamp()}"
  }
```

How do you create a Terraform resource?

For the purpose of this article, we’re going to use the Azure provider.

Step 1: Configure the provider

Configure the Azure provider in your configuration file. This enables you to specify the Azure authentication details either through the Azure CLI or using a Service Principal.

```
provider "azurerm" {
  features {}
}
```

Step 2: Find the documentation page for your resource

Step 3: Add the resource code block in the configuration file

Add the resource code block in your configuration file as per the example on the Terraform docs page, changing values where required and adding in any additional attributes from the attributes reference section:

```
resource "azurerm_resource_group" "example" {
  name     = "example"
  location = "West Europe"
  tags     = "rg1"
}
```

Step 4: Initialize and apply the configuration

Initialize and apply the configuration, accepting the planned changes when prompted:

```
terraform init
terraform apply
```

Step 5: Verify the configuration

Verify the resource group has been created successfully. You could run an Azure CLI command to do this:

Terraform resources custom condition checks

Custom condition checks within your configuration can be used to conditionally create or configure resources based on specific criteria or conditions.

- `count` — Using the count argument, you can specify a boolean value to determine whether a resource is created or not. In the example below, if the value of `var.create_instance` is true (1), then the resource is created, if false (0), it is not.

```
resource "aws_instance" "example" {
  count = var.create_instance ? 1 : 0

  ami           = "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}
```

- `condition ? true_val : false_val` can be used to set resource attributes conditionally. In this example, the `ami` attribute is set based on the value of the `use_custom_ami` variable.

```
resource "aws_instance" "example" {
  ami           = var.use_custom_ami ? "ami-abcdef12345" : "ami-0123456789abcdef0"
  instance_type = "t2.micro"
}
```

Terraform resources operation timeouts

Operation timeouts refer to the maximum amount of time Terraform will wait for a specific resource operation (e.g., creation, modification, or deletion of a resource) to complete before considering it a failure.

The available timeouts for each resource are shown on its Terraform docs page.

For example, below we set the timeouts for our resource group:

```
resource "azurerm_resource_group" "example" {
  name     = "example"
  location = "West Europe"
  tags     = "rg1"

  timeouts {
```

```
    delete = "10m"
  }
}
```

Terraform periodically checks the status of resource operations based on a polling interval, which is usually a few seconds. This polling interval is not configurable by users. Terraform continues checking the resource's status until it either succeeds, reaches the specified timeout, or encounters an error.

Terraform resources best practices

Here are a few best practices you might want to consider when creating your resources:

1. Organize your Terraform code into reusable modules to promote code reusability and maintainability. Modules will contain one or more resources.
2. Avoid Hardcoding Values in your resource blocks. Use variables and data sources to fetch dynamic information, like AMI IDs or IP addresses.
3. Limit the use of conditional logic in your configurations. It can make the code harder to understand and maintain. I prefer to use module input variables for flexibility.
4. Follow a consistent naming convention for resources making it easier to identify and manage resources, especially in large deployments.
5. Periodically review your infrastructure for unused or deprecated resources. Remove or de-provision resources that are no longer needed.

Key points

meta-arguments to configure it. Understanding how to use each part of the resource documentation is key to using Terraform, and knowing which meta-arguments are available for use can power up your deployments!

We encourage you also to explore how Spacelift makes it easy to work with Terraform. If you need any help managing your Terraform infrastructure, building more complex workflows based on Terraform, and managing AWS credentials per run, instead of using a static pair on your local machine, Spacelift is a fantastic tool for this. It supports Git workflows, policy as code, programmatic configuration, context sharing, drift detection, and many more great features right out of the box. You can check it for free, by creating a trial account.

Note: New versions of Terraform will be placed under the BUSL license, but everything created before version 1.5.x stays open-source. OpenTofu is an open-source version of Terraform that will expand on Terraform's existing concepts and offerings. It is a viable alternative to HashiCorp's Terraform, being forked from Terraform version 1.5.6. OpenTofu retained all the features and functionalities that had made Terraform popular among developers while also introducing improvements and enhancements. OpenTofu is the future of the Terraform ecosystem, and having a truly open-source project to support all your IaC needs is the main priority.

Written by

Jack Roper

Jack Roper is a highly experienced IT professional with close to 20 years of experience, focused on cloud and DevOps technologies. He specializes in Terraform, Azure, Azure DevOps, and Kubernetes and holds multiple certifications from Microsoft, Amazon, and Hashicorp. Jack enjoys writing technical articles for well-regarded websites.

Read also

TERRAFORM

23 min read

What is tfsec? How to Install, Config, Ignore Checks

TERRAFORM

9 min read

Terraform Custom Conditions – Preconditions & Postconditions