

Terraform Dynamic Blocks

Explore the chapters: 3 – Terraform Dynamic Blocks

Terraform is a popular [Infrastructure as Code \(IaC\)](#) tool among DevOps teams because it is a simple and scalable framework for developing and deploying infrastructure across many cloud providers. Terraform dynamic blocks are particularly useful in reducing redundancy in IaC code and streamlining the creation of multiple similar resources.

In this article, to help you get started with Terraform dynamic blocks, we'll take a closer look at their benefits, how they work, and walk through some practical examples so you can get hands-on with dynamic blocks.

Key benefits of Terraform

Before we jump into dynamic blocks, let's review the benefits of Terraform as a whole. There is no shortage of IaC tools and platforms available to DevOps teams. Many decide to use Terraform because of these key benefits:

- **Open-source** – developers are free to build their own providers and modules, making it platform agnostic rather than focusing on a single provider, for example, AWS CloudFormation and Microsoft's ARM templates for Azure.
- **Declarative** – developers declare the *desired* state of the infrastructure in manifest files and modules, while Terraform creates and maintains a separate *current* state file. Deploying changes is simple by amending the *desired* state, and developers can re-use code to build identical environments.
- **Planning** – Terraform gives an ability to plan deployments, allowing integration into the automated testing routines within a continuous deployment pipeline.
- **Speed** – Terraform performs operations asynchronously, communicating directly with provider APIs. Terraform deploys infrastructure incredibly fast, especially within large environments.

What are Terraform dynamic blocks?

[Terraform dynamic blocks](#) are a special Terraform block type that provide the functionality of a [for expression](#) by creating multiple nested blocks.

The need to create identical (or similar) infrastructure resources is common. A standard use case is multiple virtual server instances on a cloud platform like AWS or Azure. Terraform provides routines such as *for_each* and *count* to simplify deploying these resources, removing the requirement for large blocks of duplicate code.

Additionally, teams may need to configure multiple duplicate elements *within* a resource. In conjunction with a *for_each* routine, dynamic blocks are used within an infrastructure resource to remove the need for multiple duplicate “blocks” of Terraform code.

Key benefits of Terraform dynamic blocks

The key benefits of Terraform dynamic blocks are:

- **Speed** – simplifying the code makes it much quicker to write and also for it to be processed and thus for the infrastructure to be deployed.
- **Clarity** – in contrast to multiple blocks of repetitive code, it's much easier to read and understand code written using dynamic blocks.
- **Re-use** – copying, pasting, and amending large blocks of code is difficult and tedious. Combine dynamic blocks and variables/parameters to streamline this process.
- **Reliability** – linked to clarity and re-use, errors are less likely to be made in simple, easy-to-read code.

Why dynamic blocks?

Within Infrastructure as Code (and programming in general), there are two established language types for writing code to deploy infrastructure:

- **Imperative / Procedural** – step by step commands written out to explicitly describe how resources are created – for example, AWS CLI and Python.
- **Declarative** – the desired state of resources is declared and the tool itself works out how to create these resources – for example, AWS CloudFormation and Terraform.

Deploying infrastructure with declarative code has several advantages, such as the speed at which teams can make changes and the ease of re-using code. A key advantage of imperative languages is the ability to repeat tasks since they have built-in routines to enable this (such as *for* and *while* loops).

To provide looping “repeat” functionality in Terraform, HashiCorp developed routines and meta-arguments within their [HashiCorp Configuration Language \(HCL\)](#). These include *for_each* and *count*, which administrators can use to deploy similar resources without declaring individual code blocks.

×

Welcome to CloudBolt! Can I help you find what you're looking for?

The below code shows one way of deploying multiple subnets within a VPC in AWS using the *for_each* meta-argument.

```
# VPC variable
variable "vpc-cidr" {
  default = "10.0.0.0/16"
}

# Subnets variable
variable "vpc-subnets" {
  default = ["10.0.0.0/20","10.0.16.0/20","10.0.32.0/20"]
}

resource "aws_vpc" "vpc" {
  cidr_block = var.vpc-cidr
}


resource "aws_subnet" "main-subnet" {
  for_each = toset(var.vpc-subnets)
  cidr_block = each.value
  vpc_id = aws_vpc.vpc.id
}
```

Example code to deploy subnets within a VPC in AWS using Terraform

These routines can dramatically simplify the code required to deploy multiple resources. For example, a large organization deploying hundreds of [EC2 instances](#) for a set of legacy applications can save hours of provisioning time.

Dynamic blocks within Terraform take this concept a step deeper. Their purpose is to create multiple similar elements *within* a resource.

You can see a clear example of this benefit when deploying [AWS Security Groups](#) or [Azure Network Security Groups](#). Security groups contain rules to describe access control lists (ACLs). To streamline security group provisioning, administrators can deploy the rules with Terraform by expressing each one in turn or by using dynamic blocks.



Hybrid Cloud Solutions Demo
See the best multi-cloud management solution on the market, and when you book & attend your CloudBolt demo we'll send you a \$100 Amazon Gift Card.

Book demo

How to create an AWS Security Group with Terraform dynamic blocks

Now let's walk through a practical example of how to deploy a security group in AWS. Let's assume we have these requirements:

- Create a security group name *webserver*.
- Allow inbound HTTP (80) and HTTPS (443) from the internet (0.0.0.0/0) for web access.
- Allow outbound access on HTTPS (443) and SQL traffic (1433) to all IP addresses within the VPC for access to the database. ⚠️ Note: this outbound access rule is very broad and not best practice, but it's useful for this example.

The standard Terraform code for deployment is below:

```
# Security Groups
resource "aws_security_group" "sg-webserver" {
  vpc_id = aws_vpc.vpc.id
  name = "webserver"
  description = "Security Group for Web Servers"

  ingress {
    protocol = "tcp"
    from_port = 80
    to_port = 80
    cidr_blocks = [ "0.0.0.0/0" ]
  }


  ingress {
    protocol = "tcp"
    from_port = 443
    to_port = 443
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  egress {
    protocol = "tcp"
    from_port = 443
    to_port = 443
  }
}
```


```
cidr_blocks = [ var.vpc-cidr ]
}

egress {
  protocol = "tcp"
  from_port = 1433
  to_port = 1433
  cidr_blocks = [ var.vpc-cidr ]
}
}
```

The above code will work, but there’s unnecessary repetition – for each *ingress* and *egress* rule, there is a duplicate block of text. Any additional rules make this duplication hard to read and reproduce.



Terraform + CloudBolt = Integrated enterprise workflows



Platform	Infra as Code (IaC)	Multi Cloud Support	Self-Service User Interface	Provisioning Approval Process	Cost Control	Integrations Like ServiceNow and Ansible
Terraform	✓	✓				
Terraform + CloudBolt	✓	✓	✓	✓	✓	✓

Don't let detractors impede enterprise-wide Terraform adoption

[Learn More](#)

Using a dynamic block for each element simplifies the issue – one dynamic block for all *ingress* rules and another for all *egress* rules:

```
locals {
  inbound_ports = [80, 443]
  outbound_ports = [443, 1433]
}

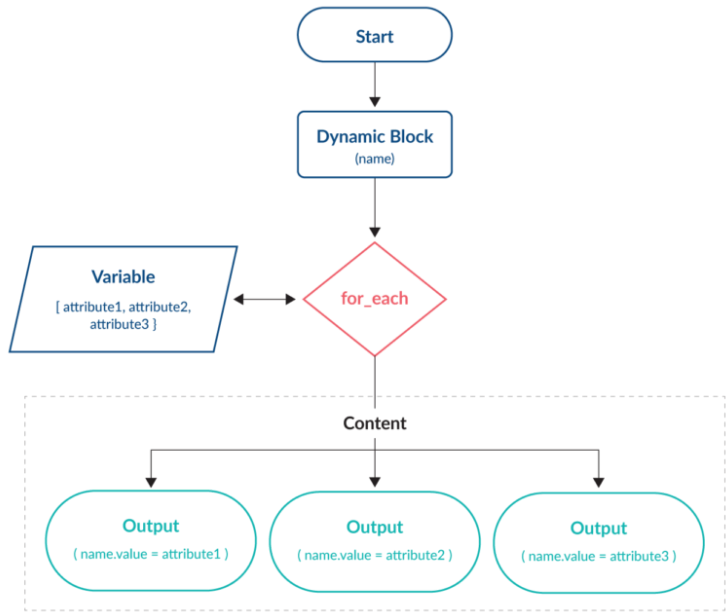
# Security Groups
resource "aws_security_group" "sg-webserver" {
  vpc_id = aws_vpc.vpc.id
  name = "webserver"
  description = "Security Group for Web Servers"

  dynamic "ingress" {
    for_each = local.inbound_ports
    content {
      from_port = ingress.value
      to_port = ingress.value
      protocol = "tcp"
      cidr_blocks = [ "0.0.0.0/0" ]
    }
  }

  dynamic "egress" {
    for_each = local.outbound_ports
    content {
      from_port = egress.value
      to_port = egress.value
      protocol = "tcp"
      cidr_blocks = [ var.vpc-cidr ]
    }
  }
}
```

There is now only a single block of dynamic code for each ingress and egress element. A *for_each* loop iterates between the local variables’ values (inbound_ports and outbound_ports). The content section contains the code from the original standard block, but the from_port and to_port attributes are replaced with those listed in the local variables.

The diagram below shows how the dynamic block uses the local variable, the *for_each* loop routine, and the *content* code block to create each output:



Dynamic block flowchart using an input variable.

The key parts of the code are highlighted below for the *inbound_ports* local variable and the *ingress* dynamic block:

```
locals {
  inbound_ports = [80, 443]
  ...
}
...

dynamic "ingress" {
  for_each = local.inbound_ports
  content {
    from_port = ingress.value
    to_port   = ingress.value
    protocol  = "tcp"
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}
```

The dynamic *ingress* block replaces all the previous duplicated *ingress* blocks. Each entry in the local *inbound_ports* variable is assigned to the *ingress.value* attribute on each iteration.

With two entries stored within the local *inbound_ports* variable (80 and 443), there will be two iterations and thus a rule for each port. To add further ingress ports, simply add a new entry in the local *inbound_ports* variable.



"CloudBolt allows us to move faster with Terraform than previously with Terraform alone"

Head of Cloud Engineering & Infrastructure Global Entertainment Company

Watch 2 minute Video

How to use Terraform dynamic blocks with lists

The code above works well for our first example as we only changed one attribute in each dynamic block (the port associated with the security group rule). Suppose a new requirement to allow outbound rules to declare different destinations on different ports. We can achieve this by amending the local "outbound" variable to contain a list:

```
locals {
  db_instance = "10.0.32.50/32"
  inbound_ports = [80, 443]
  outbound_rules = [{
    port = 443,
    cidr_blocks = [ var.vpc-cidr ]
  }, {
    port = 1433,
    cidr_blocks = [ local.db_instance ]
  }]
}

# Security Groups
resource "aws_security_group" "sg-webserver" {
  vpc_id      = aws_vpc.vpc.id
  name        = "webserver"
  description = "Security Group for Web Servers"
```

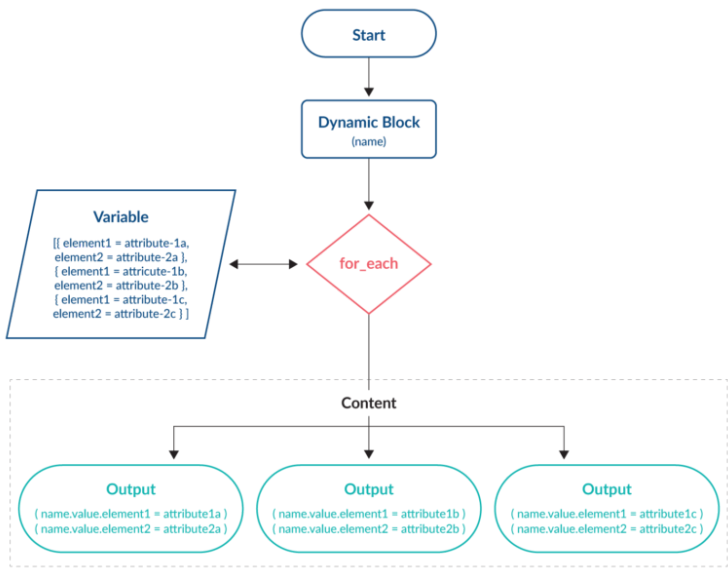
```
dynamic "ingress" {
  for_each = local.inbound_ports
  content {
    from_port = ingress.value
    to_port   = ingress.value
    protocol  = "tcp"
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}

dynamic "egress" {
  for_each = local.outbound_rules
  content {
    from_port = egress.value.port
    to_port   = egress.value.port
    protocol  = "tcp"
    cidr_blocks = egress.value.cidr_blocks
  }
}
```

Here, the local *outbound_rules* variable now contains a *list* of attributes including the port and the destination CIDR block. This configuration allows different values for these for each *egress* rule.

The *content* section contains the same code, but now the *from_port*, *to_port* and the *cidr_blocks* attributes are replaced with those listed in the local *outbound_rules* variable.

This diagram shows the effect changing the local variable to a list has on the *content* block and the output of the *for_each* loop:



Dynamic Block flowchart using an input variable containing a list.

Within the code, the key parts are highlighted below for the *outbound_rules* local variable and the *egress* dynamic block:

```
locals {
  db_instance = "10.0.32.50/32"
  outbound_rules = [{
    port = 443,
    cidr_blocks = [ var.vpc-cidr ]
  },{
    port = 1433,
    cidr_blocks = [ local.db_instance ]
  }]
}

...

dynamic "egress" {
  for_each = local.outbound_rules
  content {
    from_port = egress.value.port
    to_port   = egress.value.port
    protocol  = "tcp"
    cidr_blocks = egress.value.cidr_blocks
  }
}
```

The dynamic *egress* block replaces all the previous duplicated *egress* blocks. The entries in the local *outbound_rules* variable are assigned to the *egress.value.port* and *egress.value.cidr_blocks* attributes on each iteration.

With two lists stored within the local *outbound_rules* variable, there will be two iterations and thus a rule for each port and *cidr_block* entry. As with the first example, to add further egress rules, simply add a new list entry in the local *outbound_rules* variable.

The previous examples are simple and designed to demonstrate how dynamic blocks work, but real-world scenarios can be more complex.

For example, an administrator may need to limit Active Directory (AD) traffic inbound to Domain Controllers. AD requires multiple, specific ports to operate. Dynamic blocks can simplify the Terraform code and make it easier to read, update, and re-use in the future.

💡 **Pro-tip:** in Terraform, where both *tcp* and *udp* protocol types are required, use “-1”.

```
locals {
  inbound_rules = [
    { port = 53, protocol = "-1", # DNS
    { port = 88, protocol = "-1", # Kerberos
    { port = 123, protocol = "udp", # Time Sync (NTP)
    { port = 135, protocol = "tcp", # RPC Endpoint Mapper
    { port = 389, protocol = "-1", # LDAP
    { port = 445, protocol = "tcp", # SMB
    { port = 464, protocol = "-1", # Kerberos (password)
    { port = 636, protocol = "tcp", # LDAP SSL
    { port = 3268, protocol = "tcp", # LDAP Global Catalog
    { port = 3269, protocol = "tcp" } # LDAP Global Catalog SSL
  ]
}

# Security Groups
resource "aws_security_group" "sg-ad" {
  vpc_id      = aws_vpc.vpc.id
  name        = "active_directory"
  description = "Security Group for AD Domain Controllers"

  dynamic "ingress" {
    for_each = local.inbound_rules
    content {
      from_port = ingress.value.port
      to_port   = ingress.value.port
      protocol  = ingress.value.protocol
      cidr_blocks = var.vpc-cidr
    }
  }

  tags = {
    Name = "sg-webserver",
    Environment = var.aws_environment
  }
}
```

Here are a few more examples to help demonstrate the value of dynamic blocks.

- **Applying AWS tags** – Many large corporations have advanced tagging requirements for cloud resources to monitor and manage their environments. These are especially important when it comes to monitoring costs. When deploying new EC2 instances using an auto-scaling group within AWS, tags must be set with a *propagate_at_launch* This prevents these tags from using the common *tags* variable. Use a single dynamic block to iterate on the *tags* variable and set setting the *propagate_at_launch* attribute to ensure they are pushed to all deployed EC2 instances.

```
common_tags = {
  Name = "ddt_webapp_asg-dev"
  Environment = "dev"
  Power = "auto"
  CostCentre = "100123"
  Department = "marketing"
  Project = "new-campaign"
  Deployment = "terraform"
}
```

...

```
dynamic "tag" {
  for_each = local.common_tags
```

Subscribe for more content like this!

Email

```
content {
  key = tag.key
  value = tag.value
  propagate_at_launch = true
}
}
```

- **Creating Azure subnets** – Within Azure, it’s possible to create subnets as elements in-line within the *azurerm_virtual_network* Dynamic blocks can be combined with the *zipmap* function to build multiple subnets from two variables with minimal repetitive code, allowing for easier future deployments.

```
# Azure Subnets
variable "vnet-cidr" {
  default = "10.0.0.0/16"
}

variable "subnet_names" {
  description = "A list of public subnets inside the vNet."
  type        = list(string)
  default     = ["tdb-subnet-pub","tdb-subnet-pri"]
}

variable "subnet_prefixes" {
  description = "The address prefix to use for the subnet."
  type        = list(string)
  default     = ["10.0.0.0/20","10.0.16.0/20"]
}

resource "azurerm_resource_group" "vnet-rg" {
  name     = "test-dyn-block-rg"
  location = "uksouth"
}

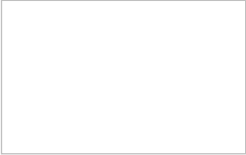
resource "azurerm_virtual_network" "vnet" {
  name                = "test-dyn-block-vnet"
  resource_group_name = azurerm_resource_group.vnet-rg.name
  location            = "uksouth"
  address_space       = var.vnet-cidr
  dynamic "subnet" {
    for_each = zipmap(var.subnet_names,var.subnet_prefixes)
    content {
      name = subnet.key
      address_prefix = subnet.value
    }
  }
}
```

- **Deploying VMware virtual machines with multiple virtual disks** – Terraform can be used to deploy virtual machines within VMware vSphere / vCloud. There is sometimes a need to deploy multiple virtual disks to a virtual machine. Terraform dynamic blocks can simplify the code and make adding additional disks easier.

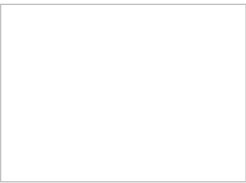
```
locals {
  disk_profiles = [{
    label = "disk0",
    size = 30
  },{
    label = "disk1",
    size = 100,
  }]
}


...

dynamic "disk" {
  for_each = local.disk_profiles
  content {
    label = disk.value.label
    size  = disk.value.size
    thin_provisioned = true
  }
}
```




Terraform + CloudBolt = Integrated enterprise workflows






Allow less technical users launch your Terraform scripts from a user interface



Let managers approve provisioning via workflows and 3rd-party integrations



Don't allow the lack of cost reporting get in the way of Terraform's adoption

Don't let detractors impede enterprise-wide Terraform adoption

Learn More

Conclusion

Dynamic blocks are a great way to simplify a set of repeating elements within a Terraform resource where it's sensible to do so.

However, dynamic blocks are not always the correct answer. You should not shift duplication and complexity from one area to another by defining all the attributes of each element of a dynamic block in the variables.

As with any new technique, apply standard common-sense principles during the development and review of the code. If only a few nested block attributes vary through each iteration, dynamic blocks are usually the right choice for deploying duplicate elements within a resource.

Explore the Chapters

Introduction – Terraform Best Practices

1 – Terraform Lookup

2 – Terraform vs. Ansible

3 – Terraform Dynamic Blocks

4 – Terraform For Loop

5 – Terraform Template

6 – Terraform Commands

7 – Terraform Tags

8 – Terraform Functions

9 – Terraform Screenshots: Practical Examples

10 – Terraform Import Examples

11 – Pulumi vs. Terraform

[Produced in partnership with Inbound Square](#)



Contact Support

Contact Sales

© 2023 CloudBolt Software, Inc.
All Rights Reserved.

<u>SOLUTIONS</u>	<u>RESOURCES</u>	<u>PROOF</u>	<u>COMPANY</u>
Why CloudBolt?	Resource Center	Did You Know?	About CloudBolt
Enterprise	Case Studies	Case Studies	Leadership
Service Provider	Blog	Make the Case	Partners
Use Cases	Research	<u>FOR MSPs / CSPs</u>	Rainmaker Partner Program
Make Better	Compare		Our Customers
Framework	CloudBolt University		Support
			Careers
			News
			Events
			Contact Us
			Dig Deeper