**Resource Dependencies**

**Introduction**

In this lesson, we're going to introduce resource dependencies, where we'll not only see a configuration with multiple resources for the first time, but also scenarios where resource parameters use information from other resources.

Up to this point, our example has only contained *a single resource*. Real infrastructure has a diverse set of resources and resource types. Terraform configurations can contain *multiple resources*, multiple resource types, and these types can even span multiple providers.

In this lesson, we'll show a basic example of multiple resources and how to reference the attributes of other resources to configure subsequent resources.

**Note:**

You should have completed the previous guides in this track, or use the following configuration to start this guide.

**Assigning an Elastic IP**

We'll improve our configuration by assigning an elastic IP to the EC2 instance we're managing. Modify your example.tf and **add the following to the end of the file**.

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

}

resource "aws\_eip" "ip" {

vpc = true

instance = aws\_instance.example.id

}

This should look familiar from the earlier example of adding an EC2 instance resource, except this time we're building an "aws\_eip" resource type. This resource type allocates and associates an elastic IP to an EC2 instance.

The only parameter for aws\_eip is "instance" which is the EC2 instance to assign the IP to. For this value, we use an interpolation to use an attribute from the EC2 instance we managed earlier.

The syntax for this interpolation should be straightforward: it requests the "id" attribute from the "aws\_instance.example" resource.

### Apply Changes

Run terraform apply to see how Terraform plans to apply this change. The output will look similar to the following:

$ terraform apply

# ...

# aws\_eip.ip will be created

+ resource "aws\_eip" "ip" {

+ allocation\_id = (known after apply)

+ association\_id = (known after apply)

+ domain = (known after apply)

+ id = (known after apply)

+ instance = (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)

+ vpc = (known after apply)

}

# aws\_instance.example will be created

+ resource "aws\_instance" "example" {

+ ami = "ami-b374d5a5"

# ...

Terraform will create two resources: the instance and the elastic IP. In the "instance" value for the "aws\_eip", you can see the raw interpolation is still present. This is because this variable won't be known until the "aws\_instance" is created. It will be replaced at apply-time.

As usual, Terraform prompts for confirmation before making any changes. Answer yes to apply. The continued output will look similar to the following:

# ...

aws\_instance.example: Creating...

aws\_instance.example: Creation complete after 36s [id=i-0214dc13c9f74e01b]

aws\_eip.ip: Creating...

aws\_eip.ip: Creation complete after 2s [id=eipalloc-007abcd5e15792497]

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

As shown above, Terraform created the EC2 instance **before** creating the Elastic IP address. Due to the interpolation expression that passes the ID of the EC2 instance to the Elastic IP address, Terraform is able to infer a dependency, and knows it must create the instance first.

( **İnterpolation** : İlk kez [Uygulamalı Matematik](https://tr.wikipedia.org/w/index.php?title=Uygulamal%C4%B1_Matematik&action=edit&redlink=1) biliminin bir alt kategorisi olan [Sayısal Analiz](https://tr.wikipedia.org/wiki/Say%C4%B1sal_Analiz) yöntemlerinde tanımlanan ve elde varolan (bilinen) değer [noktalarından](https://tr.wikipedia.org/wiki/Nokta) yola çıkarak bu noktalar arasında, farklı bir yerde ve değeri bilinmeyen bir noktadaki olası değeri bulmaya/tahmin etmeye yarayan yöntemlerin tümüne verilen genel isimdir.[[1]](https://tr.wikipedia.org/wiki/%C4%B0nterpolasyon#cite_note-Uygulamal%C4%B1%C4%B0statistik-1) En basit tanımı ile "varolan sayısal değerleri kullanarak, boş noktalardaki değerlerin tahmin edilmesi" olarak açıklanmaktadır. [Türkçede](https://tr.wikipedia.org/wiki/T%C3%BCrk%C3%A7e) bazen kolaylık olsun diye "**interpolasyon**" sözcüğü yerine yalnızca "**tahmin**" de kullanılmaktadır.

İnterpolasyon genelde [mühendislik](https://tr.wikipedia.org/wiki/M%C3%BChendislik) ve [deneylere](https://tr.wikipedia.org/wiki/Deney)/ölçümlere dayalı benzeri bilim dallarında, toplanan verilerin bir fonksiyon eğrisine **uydurulması** amacıyla kullanılmaktadır.[[2]](https://tr.wikipedia.org/wiki/%C4%B0nterpolasyon#cite_note-VeriAnalizi-2) Elde toplanan verinin dağınık ve özellikle aşırı [heterojen](https://tr.wikipedia.org/wiki/Heterojen) olduğu durumlarda interpolasyon ile boş alanlardaki değerlerin bulunması önem kazanmaktadır.

[Ekstrapolasyon](https://tr.wikipedia.org/wiki/Ekstrapolasyon) bilinen noktaların dışındaki bir alanda da tahmin yapmak için kullanılır. Kaynak : Wikipedia)

**Implicit and Explicit Dependencies**

By studying the resource attributes used in interpolation expressions, Terraform can automatically infer(çıkarım yapar) when one resource depends on another. In our example, the reference to aws\_instance.example.id creates an *implicit dependency* on the aws\_instance named example.

Terraform uses this dependency information to determine the correct order in which to create the different resources. In our example, Terraform knows that the *aws\_instance must be created before the aws\_eip*.

**Implicit dependencies** via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

Sometimes there are dependencies between resources that are not visible to Terraform. The depends\_on argument is accepted by any resource and accepts a list of resources to create *explicit dependencies* for.

*For example,* *perhaps an application we will run on our EC2 instance expects to use a specific Amazon S3 bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use depends\_on to explicitly declare the dependency:*

# New resource for the S3 bucket our application will use.

resource "aws\_s3\_bucket" "example" {

# NOTE: S3 bucket names must be unique across \_all\_ AWS accounts, so

# this name must be changed before applying this example to avoid naming

# conflicts.

bucket = "clarusway-terraform-guide"

acl = "private"

}

# Change the aws\_instance we declared earlier to now include "depends\_on"

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

# Tells Terraform that this EC2 instance must be created only after the

# S3 bucket has been created.

depends\_on = [aws\_s3\_bucket.example]

}

**Non-Dependent Resources**

We can continue to build this configuration by adding another EC2 instance:

resource "aws\_instance" "another" {

ami = "ami-b374d5a5"

instance\_type = "t2.micro"

}

Because this new instance does not depend on any other resource, it can be created in parallel with the other resources. Where possible, Terraform will perform operations concurrently to reduce the total time taken to apply changes.

Before moving on, remove this new resource from your configuration and run **terraform apply** again to destroy it. We won't use this second instance any further in the Terraform course.