

## Terraform'a Giriş — #1



Terraform, HashiCorp tarafından piyasaya sürülen bir açık kaynak “infrastructure as a code” (IaC) tool’udur. Bu tool ile bulut sistemlerde birçok servisi aynı anda kurmak/yönetmek ve birçok defa kullanmak mümkündür.

Terraform; Açık kaynaktır, **Stateful**’dur, Declarative’dir, AWS, Azure, GCP, GitHub, Docker gibi birçok provider ile uyumludur, Ücretsizdir.

### **Terraform’un Avantajları:**

#### **Platform Agnostic**

Birçok bulut sistem ve platform Terraform ile aynı anda yönetilebilir.

#### **State management**

Tüm resource’lar state file ile elimizin altında olur.

#### **Operator Confidence**

Kod yazıldıktan sonra operatöre yapıyı incelemesi istenir ve gözden kaçan hatalar ayıklanabilir.

### **Terraform Nasıl Yüklenir**

<https://www.terraform.io/downloads.htm>

#### **Ubuntu/Debian**

```
$ curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -  
$ sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com  
$(lsb_release -cs) main"  
$ sudo apt-get update && sudo apt-get install terraform
```

#### **Centos/RHEL**

```
$ sudo yum install -y yum-utils  
$ sudo yum-config-manager --add-repo
```

```
https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
```

```
$ sudo yum -y install terraform
```

## MacOS

```
$ wget
```

```
https://releases.hashicorp.com/terraform/1.1.0/terraform_1.1.0_linux_amd64.zip
```

```
$ unzip terraform_1.1.0_linux_amd64.zip
```

Unzip'lediğiniz dosyayı yeni oluşturduğunuz bir klasörün içine de atabilirsiniz.

```
$ mkdir terraform_sample
```

```
$ mv terraform terraform_sample/
```

Son olarak da herhangi bir editör ile (vim veya nano gibi) .profile dosyasını açarak bash profilini düzenleyin. Aşağıdaki komutu dosyanın sonuna ekleyin ve kaydedip çıkın.

```
export PATH="$PATH:~/terraform_sample"
```

Önemli not:

.profile dosyası gizlidir ve ls -la ile aranmalıdır.

PATH değişkenini güncellemek için `source ~/.profile` komutunun çalıştırılması gerekebilir.

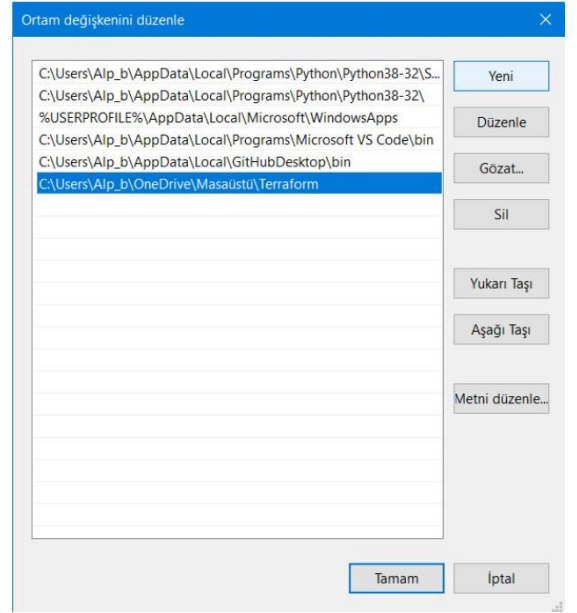
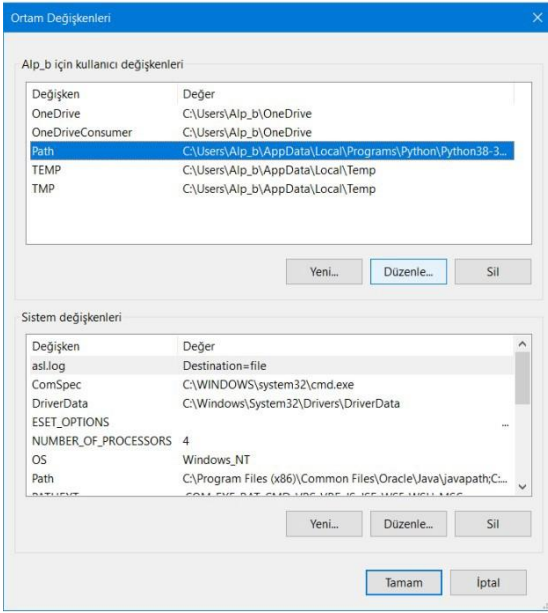
İşlemler yapıldıktan sonra `***$ terraform version***` komutuyla yüklemeyi test edin.

## Windows

Dosyayı yükledikten sonra **.zip** klasörünün içindeki terraform dosyasını dışarı çıkarın ve bu dosyanın yolunu **PATH**'inize ekleyin.

Bunun için

1. Bilgisayarım / Bu Bilgisayar'ı açıp boşlukta sağ tıklayın,
2. Sistem penceresinde sol sütunda "Gelişmiş sistem ayarları"nı seçin,
3. Sistem Özellikleri penceresinde "Gelişmiş" / "Ortam Değişkenleri" sekmesini seçin,
4. Path'inizi düzenleyerek Terraform.exe dosyanızın bulunduğu klasörün yolunu ekleyin.



Cmd veya PowerShell'de **terraform version** yazarak kontrolü sağlayın.

## Amazon Linux

```
$ sudo yum update -y
```

```
$ sudo yum install -y yum-utils
```

```
$ sudo yum-config-manager --add-repo
```

```
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
```

```
$ sudo yum -y install terraform
```

```
$ terraform version
```

\$ terraform \$ terraform -help komutları ile başlıca terraform komutlarını listeleyebilirsiniz.

```
$ terraform -help apply ya da $ terraform apply -help
```

```
$ terraform -help plan ya da $ terraform plan -help gibi komutlar ile alt komutlar hakkında detaylara ulaşabilirsiniz.
```

## VS Code kullananlar için HashiCorp Terraform, IntelliJ IDEA

**Keybindings, Terraform doc snippets** eklentileri çok kullanışlı olacaktır.

Linux tabanlı işletim sistemlerinde terraform -install-autocomplete komutuyla otomatik kod tamamlamayı yükleyebilirsiniz. Böylece ter yazıp **tab'a** bastıktan sonra terraform tamamlanacaktır.

## EC2 Üzerinden Terraform Çalışmak

Çalışmaya başlamak için öncelikle uygulamamız gereken bazı hususlar var.

Bunlardan biri aws configure komutu ile access\_key ve secret\_key girerek erişim sağlamak, ya da kurduğumuz EC2'ya IAM role oluşturmak.

Secret\_key açık kaynak olarak yazıldığında bir güvenlik açığı oluşturacağından IAM role oluşturmak daha güvenli olacaktır.

Bunun için kurduğumuz EC2'yu seçerek “Actions >> Security >> Modify IAM role” menüsü altından oluşturduğumuz “AmazonEC2FullAccess” tanımlı role’ü seçiyoruz. (Daha sonra kullanılacak servisler için de erişim izinleri verilmelidir)

## Resources

Resource’lar Terraform dilinin en önemli elementidir. Infrastructure’ı oluşturan konfigure ettiğimiz objeleri (VM, DNS records vs.) tanımlar.

```
resource "aws_instance" "web" {  
    ami           = "ami-a1b2c3d4"  
    instance_type = "t2.micro"  
}
```

**Resource type:** “aws\_instance”

**Resource/local name:** “web”

**Resource name** aynı Terraform modülünde başka bir yerden bu kaynağa atıfta bulunmak için kullanılır, ancak bu modülün kapsamı dışında hiçbir önemi yoktur. **Harf veya alt çizgi ile başlamalıdır ve yalnızca harf, rakam, alt çizgi ve tire içerebilir.**

**Configuration Arguments** iki süslü parantezin ( { } ) arasına yazılanlardır.

## Providers

Providers, Terraform’un bulut sağlayıcılar, API’lar ver SaaS sağlayıcılar ile iletişime geçmesini sağlayan “plug-in”lerdir

Terraform konfigürasyonu kurulurken provider’lar mutlaka yazılmalıdır. Çünkü bu şekilde Terraform ilgili yüklemeyi yapar ve kullanır.



kubernetes



Nomad



openstack



PostgreSQL



Bitbucket



docker



CONSUL



GitHub



New Relic



DATADOG

dnsmadeeasy



vmware  
vSphere

## En Çok Kullanılan Providers

Provider blogunu Terraform'un sitesinden alabiliriz. Ya da daha kolayı için google'a terraform aws instance yazdığınızda ilk link sizi ilgili sayfaya yönlendirecektir.

**Terraform Registry** Search Providers and Modules Browse Publish Sign-in

Providers / hashicorp / aws / Version 3.69.0 Latest Version

**aws** Overview Documentation **USE PROVIDER**

**aws provider**

- > Guides
- > ACM
- > ACM PCA
- > API Gateway (REST APIs)
- > API Gateway v2 (Websocket and HTTP APIs)
- > Access Analyzer
- > Account
- > Amazon Managed Service for Prometheus (AMP)
- > Amplify Console
- > App Runner
- > AppConfig

**Resource: aws\_instance**

Provides an EC2 instance resource. This allows instances to be created, updated, and deleted. Instances also support provisioning.

**Example Usage**

**Basic Example Using AMI Lookup**

```
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"
  }
}
```

**How to use this provider**

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

**Terraform 0.13+**

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~>3.69.0"
    }
  }
}

provider "aws" {
  # Configuration options
}
```

## State File

terraform apply komutu ile **terraform.tfstate** dosyası oluşur ve içine infrastructure'ın metadata'sı kaydedilir. Bundan sonra infrastructure'da yapılacak

her değişiklik state file ile kıyaslanır ve üzerinde yapılan değişiklikler planı görmek isteyeneye ya da apply edene gösterilir.

terraform refresh real-world infrastructure'da yapılan

değişiklikleri **apply** komutunu uygulamadan state dosyasında günceller.

## Outputs

Output değerleri, infrastructure hakkındaki bilgileri **command**

**line'da** kullanılabilir hale getirir ve diğer Terraform konfigürasyonlarının

kullanması için bilgileri açığa çıkarır.

```
output "tf_example_public_ip" {
  value = aws_instance.tf-ec2.public_ip
}
output "tf_example_private_ip" {
  value = aws_instance.tf-ec2.private_ip
}
output "tf_example_s3_meta" {
  value = aws_s3_bucket.tf-s3.region
}
```

```
tf_example_private_ip = "172.31.88.164"
tf_example_public_ip = "3.92.197.0"
tf_example_s3_meta = "us-east-1"
```

Yazdığımız kod blokları ile infrastructure oluşturulduktan sonra çıktı olarak kurduğumuz EC2'nun public/private IP'leri ve S3 region yazacaktır.

## Bir Infrastructure'ın Ömrü

**Init** → Yazdığımız .tf dosyalarını öncelikle initialize etmek gerekir. Bunun için terraform init komutu kullanılır ve böylece kullanılacak providerlar yüklenir ve kullanıma hazır hale gelir.

- Provider eklentileri yüklenir.
- .terraform.lock.hcl dosyası yüklenir. Versiyon kilitlenir, böylece başkası başka versiyon ile dosya üzerinde çalışamaz.

**Plan** → 0.11 versiyonundan sonra zorunluluğu kalkan bir aşamadır. terraform plan komutu ile **current state** ve **desired state** arasında fark olup olmadığı gözlemlenir. Özellikle çoklu çalışmalarda kontrolü sağlamak için hayati önem taşır.

“+” → oluşturulacak

“-” → silinecek

“~” → değiştirilecek

- **Real-world infrastructure** içinden bazı resource bloklarını alarak yeni bir plan oluşturmak için terraform plan out <plan\_ismi> kodu girilir ve yeni bir plan oluşturulur. Çok kullanılan bir uygulama değildir.

**Apply** → Infrastructure'ı aktif hale getirmek için terraform apply komutu kullanılır.

- Execution plan tekrar gösterilir ve onay istenir.
- “Yes” dindikten sonra **creating** başlar.
- **terraform.tfstate** dosyası oluşturulur. İçerisinde **resource** bilgileri yer alır. Ana dosyada değişiklik yapıldığında bu dosyayı referans alarak değişiklikleri gösterir.
- Bir kez daha apply yapıldığında bu sefer **terraform.tfstate.backup** dosyası oluşur. Bunun içinde de bir önceki **state file** yedeklenir.
- `terraform apply -auto-approve` yazarak otomatik olarak **yes** cevabını almış olur.
- `terraform apply -refresh=false` çok fazla instance açık olduğunu varsayarsak, ufak bir girdi için zaman kaybına sebep olmamak adına refresh yapmadan uygular.

**Destroy** → Her fani gibi çalışan sistemleri terminate etmek için `terraform destroy` komutu kullanılır.

- \*\*\*Konfigürasyonda servisin ana özelliklerinden birinde değişiklik yapıp tekrar **apply** edilirse eski kaynak **terminate** edilir ve infrastructure yeniden oluşturulur. Ancak **Name-tag** gibi basit değişiklikler için terminate etmeden ekleme/çıkarma yapılır.

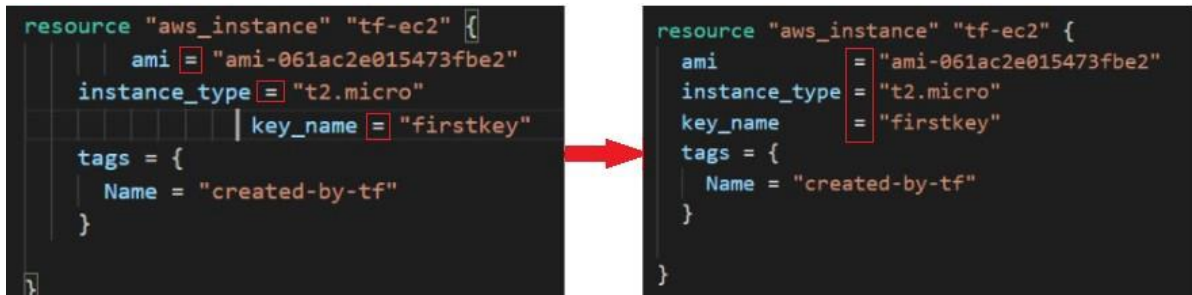
## En Çok Kullanılan Terraform Kodları

`terraform state list`

Oluşturulan tüm resource'lar listelenir. (linux **ls** komutu gibi)

`terraform fmt`

Syntax'ı düzenlemek / derleyip toplamak için kullanılır. Özellikle çok kişinin üzerinde çalıştığı kodların okunabilirliğini artırmaya yarar.



The image shows two side-by-side code snippets representing the same Terraform configuration for an AWS instance, separated by a red arrow pointing from left to right. The left snippet shows the code as it might appear before formatting, with inconsistent indentation and spacing. The right snippet shows the code after running `terraform fmt`, where the formatting is standardized: all lines are aligned to the left, and there is consistent spacing around the equals signs and curly braces.

```
resource "aws_instance" "tf-ec2" {
  ami = "ami-061ac2e015473fbe2"
  instance_type = "t2.micro"
  key_name = "firstkey"
  tags = {
    Name = "created-by-tf"
  }
}
```

`terraform validate`

Syntax hatası ve duplicate olup olmadığını gösterir. Hatalı satır ve kodu belirtir.



```
resource "aws_instance" "tf-ec2" {  
  ami           = var.ec2_ami  
  instance_type = var.ec2_type  
  key_name      = "firstkey"  
  tags = {  
    Name = "${var.ec2_name}-instance"  
  }  
}
```

**Error: Unsupported block type**

on main.tf line 18:

18: `resource` "aws\_instance" "tf-ec2" {

Blocks of type "resource" are not expected here. Did you mean "resource"?

terraform import

Real-world infrastructure'dan state dosyasına bilgi çekmeyi sağlar.

terraform output

Output bloklarındaki bilgilerin çıktısını almayı sağlar.

```
output "tf_example_s3_meta" {  
  value = aws_s3_bucket.tf-s3.region  
}
```

```
$ terraform output tf_example_s3_meta  
"us-east-1"
```

terraform graph

Konfigürasyonun ya da execution planın görsel sunumunu sağlar. Bunun için yapılması gereken **terraform graph** komutunu girdikten sonra çıkan **digraph** ile başlayan kod bloğunu kopyalamak

ve <https://dreampuf.github.io/GraphvizOnline> sitesine girip soldaki konsola yapıştırmak.

terraform show

Yazdığımız kodu **human-readable** formatta gösterir.

Machine-readable çıktı almak için sonuna `-json` eklenmelidir.

terraform console

Konsol modunu açarak işlem yapmayı sağlar.



```
$ terraform console
> min (3,5,7)
3
> lower ("DevOps")
"devops"
> aws_instance.tf-ec2.public_ip
"3.92.197.0"
>
```

**Blokların aynı dosya içinde olma şartı yoktur.** Aynı klasör içindeki tüm .tf uzantılı dosyalar beraber okunur. Böylece her blok çeşidini ayrı dosyalarda gösterip karışıklığın önüne geçilebilir.

```
main.tf x variables.tf x outputs.tf x
main.tf > resource "aws_instance" "tf-ec2" > ami
1
2 provider "aws" {
3   region = "us-east-1"
4 }
5
6 2 references
7 resource "aws_instance" "tf-ec2" {
8   ami           = var.ec2_ami
9   instance_type = var.ec2_type
10  key_name      = "firstkey"
11  tags = {
12    Name = "${var.ec2_name}-instance"
13  }
14
15 1 reference
16 resource "aws_s3_bucket" "tf-s3" {
17   bucket = var.s3_bucket_name
18   acl    = "private"
19 }
20
variables.tf x
variables.tf > variable "s3_bucket_name"
1
2 1 reference
3 variable "ec2_name" {
4   default = "alp-ec2"
5 }
6
7 1 reference
8 variable "ec2_type" {
9   default = "t2.micro"
10 }
11
12 1 reference
13 variable "ec2_ami" {
14   default = "ami-0742b4e673072066f"
15 }
16
17 1 reference
18 variable "s3_bucket_name" {
19   default = "alpp-bucket"
20 }
21
outputs.tf x
outputs.tf > output "tf_example_s3_meta"
1
2 1 reference
3 output "tf_example_public_ip" {
4   value = aws_instance.tf-ec2.public_ip
5 }
6
7 1 reference
8 output "tf_example_private_ip" {
9   value = aws_instance.tf-ec2.private_ip
10 }
11
12 1 reference
13 output "tf_example_s3_meta" {
14   value = aws_s3_bucket.tf-s3.region
15 }
16 }
```

## Terraform'a Giriş — #2



### Variables

Variable'lar neden kullanılır? Başlıca sebepleri şunlardır:

- Gizli olması gereken bilgileri saklamak (AWS credentials gibi).

- Değişme ihtimali olan bilgileri rahat kullanmak (AMI'lar region'a göre değişiklik gösterir).
- Hazır variable'ları tekrar ve kolayca kullanmak.

```
1 reference
variable "ec2_name" {
  default = "alp-ec2"
}

1 reference
variable "ec2_type" {
  default = "t2.micro"
}

1 reference
variable "ec2_ami" {
  default = "ami-0742b4e673072066f"
}
```

```
2 references
resource "aws_instance" "tf-ec2" {
  ami            = var.ec2_ami
  instance_type  = var.ec2_type
  key_name       = "firstkey"
  tags = {
    Name = "${var.ec2_name}-instance"
  }
}
```

### Variables ile kurulan EC2-instance

Terraform, değişken atamalarını aşağıdaki sıraya göre önceliklendirir:

1. `-var` ve `-var-file` komutları.
2. `*.auto.tfvars` ya da `*.auto.tfvars.json` uzantılı dosyalar (alfabetik sıraya göre).
3. `terraform.tfvars.json`
4. `terraform.tfvars`
5. Environment variables
6. Default variables

**-var**

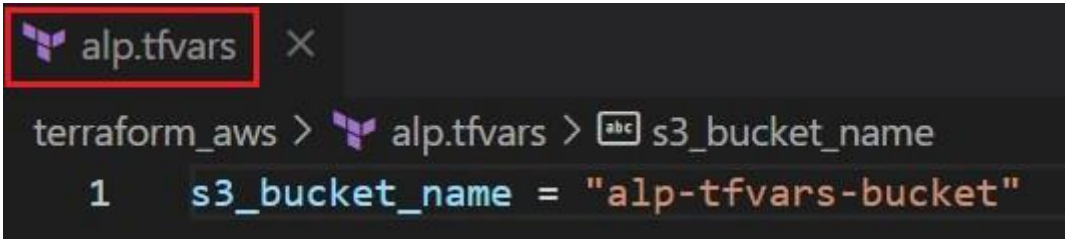
```
1 reference
variable "s3_bucket_name" {
  default = "alpp-bucket"
}
```

```
~ arn = "arn:aws:s3:::alpp-bucket" -> (known after apply)
~ bucket = "alpp-bucket" -> "alp-yeni" # forces replacement
~ bucket_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
~ bucket_regional_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
```

terraform plan -var="s3\_bucket\_name=alp-yeni" kodunu çalıştırdığımızda planda **bucket** isminin resimdeki “alpp-bucket” yerine “alp-yeni” olarak değiştirileceği görülecektir. Bunun sebebi de değişkenlerin yukarda belirtilen öncelikleridir. Uygulamak için de terraform apply -var="s3\_bucket\_name=alp-yeni" komutunu çalıştırmamız gerekmektedir.

### -var-file

Herhangi bir isimde <name>.tfvars uzantılı dosyaya değişkenleri yazdığımızda, değişkenleri uygulamak için terraform plan -var-file=<name>.tfvars yazılmalıdır.



```
alp.tfvars X
terraform_aws > alp.tfvars > s3_bucket_name
1 s3_bucket_name = "alp-tfvars-bucket"
```

\$ terraform plan -var-file="alp.tfvars"

```
~ arn = "arn:aws:s3:::alpp-bucket" -> (known after apply)
~ bucket = "alpp-bucket" -> "alp-tfvars" # forces replacement
~ bucket_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
~ bucket_regional_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
```

\*\*\*İçerisinde **terraform.tfvars** dosyası olan bir klasörde, farklı bir isimde <name>.tfvars dosyası oluşturulup -var-file eklentisi kullanılmazsa değişkenler default olarak **terraform.tfvars** dosyasından atanır.

### TF\_VAR\_

Environment variables TF\_VAR\_ komutu ile tanımlanabilir.

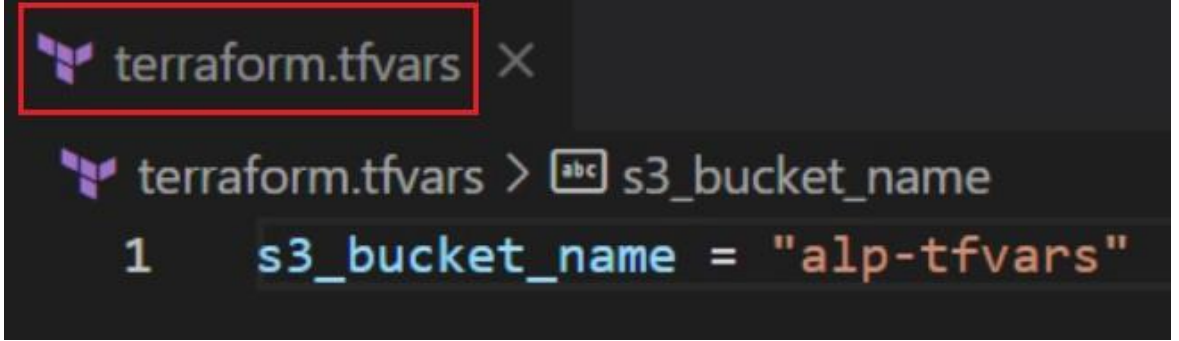
\$ export TF\_VAR\_s3\_bucket\_name=alp-env-bucket

\$ terraform plan

```
~ arn = "arn:aws:s3:::alpp-bucket" -> (known after apply)
~ bucket = "alpp-bucket" -> "alp-env-bucket" # forces replacement
~ bucket_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
~ bucket_regional_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
```

## terraform.tfvars

terraform.tfvars dosyası oluşturarak değişken atamaya bir örnek:



\$ terraform plan

```
~ arn = "arn:aws:s3:::alpp-bucket" -> (known after apply)
~ bucket = "alpp-bucket" -> "alp-tfvars" # forces replacement
~ bucket_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
~ bucket_regional_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
```

## .auto.tfvars

Farklı isimde oluşturduğumuz dosyanın öncelik kazanması için uzantısının **auto.tfvars** olması gerekmektedir.



```
~ arn = "arn:aws:s3:::alpp-bucket" -> (known after apply)
~ bucket = "alpp-bucket" -> "auto-tfvars-bucket" # forces replacement
~ bucket_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
~ bucket_regional_domain_name = "alpp-bucket.s3.amazonaws.com" -> (known after apply)
```

Kazanan **<name>.auto.tfvars**. Peki bu özellik nerede işimize yarayacak: Hazır bir şablonu birçok kişi kullanıyor ancak herkes aynı özellikleri kullanmıyor. Benim çalıştığım instance-type “t2.micro” iken başkasının “t2.large” olabilir. Bu sebeple variables ile referans edilen bu argümanlar için herkes bireysel **<name>.auto.tfvars** dosyasında kendi variable’larını belirlerse, atamalar herkesin kendine ait dosyası üzerinden gerçekleşecektir.

## Locals

Sık geçen açıklamalar ve kolay değiştirilebilir değişkenler için birçok **variable** bloğu oluşturmak yerine tek bir **locals** bloğunda birçok değişken oluşturulabilir.

```
locals {  
  mytag      = "alp-local-name"  
  ins_type   = "t2.micro"  
  ins_ami    = "ami-0742b4e673072066f"  
}  
  
resource "aws_instance" "tf-ec2" {  
  ami           = local.ins_ami  
  instance_type = local.ins_type  
  key_name      = "firstkey"  
  tags = {  
    Name = "${local.mytag}-local-named-instance"  
  }  
}
```

### Count & for\_each

Bir resource bloğu default olarak bir tane infrastructure objesi oluşturabilir. Ancak bazen ayrı bloklar yazmadan aynı objeden birden çok oluşturulması gerekebilir. `count` ve `for each` çoklu obje oluşturmak için kullanılan **meta-data** argümanlarıdır.

\*Bir resource ya da module bloğunda `count` ve `for_each` aynı anda kullanılamaz.

`count`: Her resource tipinde ve module'de kullanılabilir. Yalnızca doğal sayılar (0, 1, 2, ....) yazılabilir.

`count.index`: 0'dan başlayan dizinler için kullanılır.

`length`: String veya listenin eleman sayısını sayısal değere dönüştürmek için kullanılır.

```
variable "ec2_name" {
  type      = list(string)
  default   = ["A", "B", "CD"]
}

resource "aws_instance" "tf-ec2" {
  ami           = var.ec2_ami
  instance_type = var.ec2_type
  key_name      = "firstkey"
  count         = length(var.ec2_name)

  tags = {
    Name = "server ${count.index}"
  }
}
```

Örnekte 3 elemanlı bir liste var. `count = 3`. Yani 3 adet EC2 instance kurulacak. `count.index` yazdığımız kod (0, 1, 2) olarak atandığı için **Name-Tag'leri** server 0 / server 1 / server 2 olacaktır.

`${count.index+1}` deseydik bu sefer server 1 / server 2 / server 3 şeklinde çıktı alacaktık.

Oluşturulan instance'lar aynı ve sıralıysa `count` kullanmak kullanışlıdır. Ancak sıralı değil ve farklı adlandırmalara ihtiyaç duyuluyorsa `for each` daha kullanışlı olacaktır.

`for_each` : Her resource tipinde ve module'de kullanılabilir. Yalnızca map veya set değerler yazılabilir.

`toset` : Terraform'un set için bir syntax'ı yoktur. Ancak string listeleri `toset` ile set elemanlarına çevrilebilir.

`to-map` : Argümanları map değerine çevirir. Pek kullanılmaz.



```
resource "azurerm_resource_group" "rg" {
  for_each = {
    a_group      = "eastus"
    another_group = "westus2"
  }
  name      = each.key
  location  = each.value
}
```

## Map

```
variable "users" {
  default = ["santino", "michael", "fredo"]
}

resource "aws_iam_user" "new_user" {
  name = each.value
  for_each = toset(var.users)
}
```

## Set

Aynı anda santino, michael ve fredo isimli 3 **iam user** oluşturduk.

\***toset** - **tomap** gibi gömülü fonksiyonlar Terraform’u kullandıkça öğrenilecektir.

## Conditionals

**condition** ? **true\_value** : **false\_value** yapısı kullanılır. Bu ne demektir? Aşağıdaki örnekte **count** için eğer ki **num\_of\_buckets** sıfıra eşit değilse (yani true ise) **num\_of\_buckets** değeri yazdırılır; eşitse (false ise) “3” yazdırılır.

Variable bloğunda **num\_of\_buckets**’a 2 değeri atandığı için sıfıra eşit değildir ve **condition** doğrudur. Bu sebeple **count** değeri 2 olur.



```

variable "s3_bucket_name" {
  default = "tf-bucket"
}
variable "num_of_buckets" {
  default = 2
}
resource "aws_s3_bucket" "tf-s3" {
  bucket = "${var.s3_bucket_name}-${count.index+1}"
  acl     = "private"
  count   = var.num_of_buckets != 0 ? var.num_of_buckets : 3
}

```

Böylece **tf-bucket-1** ve **tf-bucket-2** adlı iki bucket oluşturulur.

### Loops

Birçok bucket, iam user vs. oluşturmak istiyorsak bunu **for expressions** ile kolayca yapabiliriz.

İlk kelimesi farklı isimlerden oluşan 3 adet **bucket** oluşturmak için 3 elemanlı bir liste yazıyoruz.

```

variable "users" {
  default = ["alp", "samet", "selman"]
}
resource "aws_s3_bucket" "tf-s3" {
  acl = "public-read-write"
  for_each = toset(var.users)
  bucket   = "${each.value}-s3-bucket"
}

```

Çıktı olarak **alp-s3-bucket** , **samet-s3-bucket** , **selman-s3-bucket** isimli 3 bucket oluşturulur.

İkinci örnekte ise a **variable**'ı **users** listesinin içinde **iterate** edilir ve 3 karakterden uzun olma koşulunu sağlayan elemanların büyük harfle **output**'unun alınması sağlanır.

Kısaca listedeki 3 harften uzun elemanların büyük harflerle çıktısı alınır.

```
variable "users" {  
  default = ["alp", "samet", "selman"]  
}  
  
output "uppercase_users" {  
  value = [for a in var.users : upper(a) if length(a) > 3]  
}
```

Çıktılar **SAMET** ve **SELMAN** olacaktır.

### Data Sources

Data source'u olan bir bilgiyi çekmek için kullanırız. Bu Amazon'un kendi OS'ı, snapshot'ını aldığımız bir EC2 vs. olabilir. Örneğimizi yeni bir EC2 instance ayağa kaldırma üzerinden yapacağız. Bu kod bloğu için Terraform'un zengin dokümantasyonuna ulaşmak istiyorum ve google'a "data aws\_ami terraform" yazıyorum.

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/ami>

Ulaştığım sayfada tüm detaylar hizmetime sunulmuş durumda.

```

data "aws_ami" "tf_ami" {
  most_recent      = true
  owners           = ["amazon"]

  filter {
    name = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

resource "aws_instance" "tf-ec2" {
  ami              = data.aws_ami.tf_ami.id
  instance_type    = var.ec2_type
  key_name         = "firstkey"
  tags = {
    Name = "${local.mytag}-local"
  }
}

```

AMI name  
amzn2-ami-hvm-2.0.20210326.0-x86\_64-gp2

Yaptığımız örnekte;

`most_recent true` diyerek en güncel sürümü seçtim.

`owners amazon`'un AMI'ını kullanmak istiyorum.

`filter` devreye sokarak **Root-device-type: ebs** ya da **Virtualization-type:**

**hvm** filtrelemesi de yapabilirim. Ben AMI adını kullanarak daha spesifik bir filtrelemeyi tercih ettim.

```

~ ami              = "ami-0742b4e673072066f" -> "ami-061ac2e015473fbe2" # forces replacement
~ arn              = "arn:aws:ec2:us-east-1:133446465665:instance/i-0c059c2bfecb008ea" -> (known after apply)
~ associate_public_ip_address = true -> (known after apply)
~ availability_zone = "us-east-1b" -> (known after apply)

```



Amazon Linux  
Free tier eligible

Amazon Linux 2 AMI (HVM) - Kernel 4.14, SSD Volume Type - **ami-061ac2e015473fbe2** (64-bit x86) / ami-00d06335ef617238f (64-bit Arm)

Amazon Linux 2 comes with five years support. It provides Linux kernel 4.14 tuned for optimal performance on Amazon EC2, systemd 219, GCC 7.3, Glibc 2.26, Binutils 2.29.1, and the latest software packages through extras. This AMI is the successor of the Amazon Linux AMI that is now under maintenance only mode and has been removed from this wizard.

Root device type: ebs

Virtualization type: hvm

ENA Enabled: Yes



### **Provisioners**

Terraform kendi dokümanında provisioners kullanımını son çare olarak düşünmemizi ister.

Provisioners, sunucuları veya altyapı objelerini (Infrastructure Objects) servise hazırlamak için yerel veya uzak bir makinede belirli eylemleri modellemek için kullanılır.

### **local-exec**

Terraform resource ile ayağa kaldırdığımız bir EC2'nun public ve private IP'lerini localimizde .txt dosyası içinde saklamak istiyoruz. Bunu `local-exec` komutu ile şu şekilde yaparız:

```

resource "aws_instance" "izmir" {
  count          = length(var.num)
  ami           = data.aws_ami.tf_ami.id
  instance_type = var.ins_type
  key_name       = "firstkey"
  security_groups = ["tf-provisioner-sg"]
  tags = {
    Name = "Terraform ${var.num[count.index]} Instance"
  }

  provisioner "local-exec" {
    command = "echo http://${self.public_ip} >> public_ip.txt"
  }
  provisioner "local-exec" {
    command = "echo http://${self.private_ip} >> private_ip.txt"
  }
}

```

Öncelikle 2 adet EC2 ayağa kaldırdığımızı varsayalım. Artık uzak bağlantı gereksinimi duyduğumuz için **SSH** içeren `security_groups` da tanımladık (`Sec_group` kod bloğu aşağıda paylaşılacaktır.). `provisioner` bloğunu, süslü parantezlere dikkat edersek, `resource` bloğunun içinde yazıyoruz. Bu sebeple `resource` artık **parent resource** olarak tanımlanabilir.

**provisioner** bloğunun içinde dikkatimizi çeken 2 komut var: **command** ve **self**. `command` - Tek satırlık kodları yazmak içindir. Biz burada instance'larımızın IP'lerinin .txt dosyalarının içine aktarılmasını istiyoruz.

`self` - Parent resource'un tüm niteliklerini taşıyan bir komuttur. Örnek olarak **self.public\_ip** yazarak `resource`'un public IP'sine yönlendirme yaparız. Kodumuzu **apply** ettiğimizde `infrastructure`'ı kurduğumuz klasörde **public\_ip.txt** ve **private\_ip.txt** dosyaları oluşacak, içinde de oluşturduğumuz EC2'ların IP adresleri yazacaktır.

### remote-exec

Az önce localimize yönelik bir işlem gerçekleştirdik. Şimdi `remote-exec` ile kurduğumuz EC2 içinde bir işlem yapacağız.

Örnek olarak E2'nun içinde Apache Server oluşturmak ve "Hello World" çıktısı almak istiyoruz.

```

connection {
  host      = self.public_ip
  type      = "ssh"
  user      = "ec2-user"
  private_key = file("firstkey.pem")
}

provisioner "remote-exec" {
  inline = [
    "sudo yum update -y",
    "sudo yum -y install httpd",
    "sudo systemctl enable httpd",
    "sudo systemctl start httpd",
    "echo 'Hello World' > index.html",
    "sudo cp index.html /var/www/html/"
  ]
}

```

Bu **provisioner** ve **connection** blokları da resource bloğunun içinde yer alacaktır.

host - Bağlanılacak kaynağın adresi.

type - Bağlantı tipi. Linux için ssh, Windows için winrm.

user - Amazon Linux 2 bağlantısı yaptığımız için “ec2-user”. Ubuntu olsaydı “ubuntu” olacaktı.

private\_key - key.pem dosyamız.

inline - command komutu ile tek satır yazabilirken, çoklu satır girmemizi sağlayan komut inline’dır.

**User data**’yı EC2 kurarken girmek zorundayız ancak remote-exec ile EC2 kurduktan sonra da uzaktan komut girebiliriz. Bu, ikisini birbirinden ayıran can alıcı noktadır.

Ancak sonradan eklediğimiz **provisioner** bloğunu parent resource içine eklersek Terraform herhangi bir değişiklik görmeyecektir. **Trigger** sağlamak için yeni bloğu “**null\_resource**” altında çalıştırmamız gerekecektir.

```

resource "null_resource" "izmir" {
  connection {
    host = aws_instance.instance.public_ip

```



```

type = "ssh"
user = "ec2-user"
private_key = file("firstkey.pem")
}provisioner "remote-exec" {
  inline = [
    "sudo yum -y install httpd",
    "sudo systemctl enable httpd",
    "sudo systemctl start httpd",
    "echo 'Hello World' > index.html",
    "sudo cp index.html /var/www/html/"
  ]
}
}

```

Security Group oluşturmak için gerekli kod bloğu:

```

resource "aws_security_group" "tf-sec-gr" {
  name = "tf-provisioner-sg"
  tags = {
    Name = "tf-provisioner-sg" }

  ingress {
    from_port = 80
    protocol = "tcp"
    to_port = 80
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port = 22
    protocol = "tcp"
    to_port = 22
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port = 0
    protocol = "-1"
    to_port = 0
    cidr_blocks = ["0.0.0.0/0"]
  }
}

```



```
}  
}
```

Uzak bağlantı için port-22'ye, Apache Server kurmak için port-80'e ihtiyacımız vardır.

egress bloğunda **protocol'e "-1"** yazılmasının sebebi "**all**" demektir. Yani tüm protokol çıkışlarına izin verilir.

### **Remote State**

State dosyamızı s3 bucket içine koyarak yapılan değişikliklerin tüm departmanların ulaşabileceği bir uzak bilgisayarda muhafazası sağlanabilir.. Bunun için öncelikle ayrı bir klasörde yeni bir bucket resource bloğu oluşturmamız. Bu sefer detaylandırarak versiyonlamayı açtık, destroy için zorlama talimatı verdik ve **AES256** veri şifrelemesi ekledik.

Bunların haricinde birçok kişinin ulaşabildiği bir state dosyasında bizi zora düşürecek olan durum, birden fazla kişinin aynı anda **apply** etmesi olacaktır.

Bunu engellemek için de **sınavlarda ve mülakatlarda sıkça karşımıza çıkabilecek bir konu** olan dynamoDB servisinin LockID özelliğini kullanıyoruz. `hash_key = "LockID"` yazarak bir kişinin **apply** komutunu vermesiyle yeni bir **apply** için değişikliğin bitmesi beklenecek. Aynı anda apply komutu verebilmenin önüne geçilecek.

backend.tf X

s3-backend > backend.tf > ...

```
1  provider aws {
2    region = "us-east-1"
3  }
4  resource "aws_s3_bucket" "tf-remote-state" {
5    bucket = "tf-remote-s3-bucket-alp"
6    versioning {
7      enabled = true
8    }
9    force_destroy = true
10   server_side_encryption_configuration {
11     rule {
12       apply_server_side_encryption_by_default {
13         sse_algorithm = "AES256"
14       }
15     }
16   }
17 }
18 resource "aws_dynamodb_table" "tf-remote-state-lock" {
19   hash_key = "LockID"
20   name = "tf-s3-app-lock"
21   attribute {
22     name = "LockID"
23     type = "S"
24   }
25   billing_mode = "PAY_PER_REQUEST"
26 }
```

State dosyası için yeni bir S3\_bucket oluşturduk ancak local'de çalıştığımız main.tf dosyasına da bu bilgiyi eklememiz gerekmektedir.

terraform\_aws &gt; main.tf &gt; ...

```
1  terraform {
2      required_providers {
3          aws = {
4              source = "hashicorp/aws"
5              version = "3.69.0"
6          }
7      }
8
9      backend "s3" {
10         bucket = "tf-remote-s3-bucket-alp"
11         key = "env/dev/tf-remote-backend.tfstate"
12         region = "us-east-1"
13         dynamodb_table = "tf-s3-app-lock"
14         encrypt = true
15     }
16 }
17
18 provider "aws" {
19     region = "us-east-1"
20 }
21
22 resource "aws_instance" "tf-ec2" {
23     ami           = var.ec2_ami
24     instance_type = var.ec2_type
25     key_name      = "firstkey"
26     tags = {
27         Name = "${local.mytag}-local"
28     }
29 }
```

[main.tf](#)'e eklediğimiz backend bloğu ile bundan sonra **state** dosyası local'e değil s3\_bucket içinde env/dev/tf-remote-backend.tfstate içine yüklenecektir.

## Modules

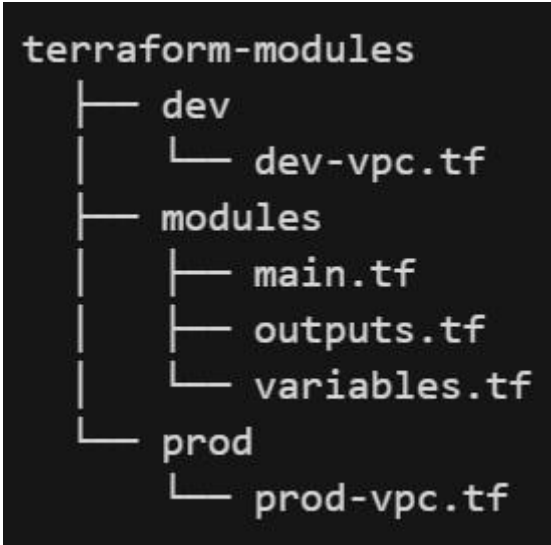
Şirketinizde 10 tane departman olduğunu düşünelim ve her departman kendi VPC'sini kurmak yerine DevOps ekibi bir tane VPC modülü oluşturur ve her departman o modülü kullanarak sistemini kurar.

Modül'ler 2'ye ayrılır:

Bir Terraform modülü (genellikle bir konfigürasyonun root modülü), kaynaklarını konfigürasyona dahil etmek için diğer modülleri çağırabilir.

1. **Root Module:** Child modülleri çağırdığımız modüle denir.
2. **Child Modules:** Root modül tarafından çağrılan modüllere denir.

Örneğimizde “Development” ve “Production” departmanlarını ele alalım. Bu iki departmana VPC modülü oluşturmak için aşağıdaki gibi bir “ağaç” oluşturalım:



**dev** ve **prod** dizinleri ile root modüllerini,  
**modules** dizini ile child modülünü oluşturmuş olduk.

```
main.tf
1 provider "aws" {
2   region = "us-east-1"
3 }
4 resource "aws_vpc" "module_vpc" {
5   cidr_block = var.vpc_cidr_block
6   tags = {
7     Name = "terraform-vpc-${var.environment}"
8   }
9 }
10 resource "aws_subnet" "public_subnet" {
11   cidr_block = var.public_subnet_cidr
12   vpc_id = aws_vpc.module_vpc.id
13   tags = {
14     Name = "terraform-public-subnet-${var.environment}"
15   }
16 }
17 resource "aws_subnet" "private_subnet" {
18   cidr_block = var.private_subnet_cidr
19   vpc_id = aws_vpc.module_vpc.id
20   tags = {
21     Name = "terraform-private-subnet-${var.environment}"
22   }
23 }
```

```
variables.tf
1 variable "environment" {
2   default = "izmir"
3 }
4
5 variable "vpc_cidr_block" {
6   default = "10.0.0.0/16"
7   description = "this is our vpc cidr block"
8 }
9
10 variable "public_subnet_cidr" {
11   default = "10.0.1.0/24"
12   description = "this is our public subnet cidr block"
13 }
14
15 variable "private_subnet_cidr" {
16   default = "10.0.2.0/24"
17   description = "this is our private subnet cidr block"
18 }
```

```
outputs.tf
1 output "vpc_id" {
2   value = aws_vpc.module_vpc.id
3 }
4
5 output "vpc_cidr" {
6   value = aws_vpc.module_vpc.cidr_block
7 }
8
9 output "public_subnet_cidr" {
10  value = aws_subnet.public_subnet.cidr_block
11 }
12
13 output "private_subnet_cidr" {
14  value = aws_subnet.private_subnet.cidr_block
15 }
```

## modules child modülü

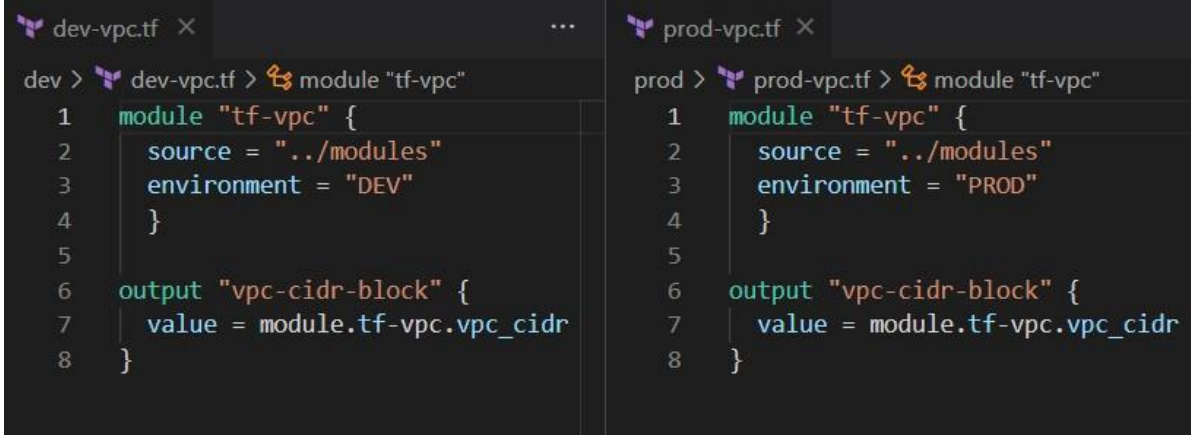
Bilgileri çekeceğimiz child modülümüzün içine baktığımızda:

10.0.0.0/16 cidr bloğu ile VPC (Name\_tag = terraform-vpc-izmir)

10.0.1.0/24 cidr bloğu ile public subnet (Name\_tag = terraform-public-subnet-izmir)

10.0.2.0/24 cidr bloğu ile private subnet (Name\_tag = terraform-private-subnet-izmir)

kurulacak.



```
dev > dev-vpc.tf > module "tf-vpc"
1  module "tf-vpc" {
2      source = "../modules"
3      environment = "DEV"
4  }
5
6  output "vpc-cidr-block" {
7      value = module.tf-vpc.vpc_cidr
8  }

prod > prod-vpc.tf > module "tf-vpc"
1  module "tf-vpc" {
2      source = "../modules"
3      environment = "PROD"
4  }
5
6  output "vpc-cidr-block" {
7      value = module.tf-vpc.vpc_cidr
8  }
```

**dev-vpc** ve **prod-vpc** adlı root modüller, **modules** klasöründen child modülü çağırarak tüm kaynaklarına erişmiş olacak.

## Import

Elimizde hazır çalışan bir kaynak var ve biz bu kaynağın bilgilerini kendi kodumuza çekmek istiyoruz. **Import** komutu bu konuda bize yardımcı olacaktır. Ancak şöyle bir durum mevcut. Hazır kaynağı bizim kodumuzun içine değil, state dosyasının içine atacaktır. Yani biz kaynak bilgilerini state dosyasından alıp kodumuzun içine işlemedikçe kaynakları kullanmamız mümkün olmayacaktır. Bu konuyla alakalı olarak doküman der ki: **“A future version of Terraform will also generate configuration.”**

Örnek olarak hazırda çalışan bir EC2-instance’ımız olsun ve biz bunu kodumuzun içine çekmek istiyoruz. Öncelikle Terraform’da bir **aws\_instance** resource’u oluşturalım:

```

provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "import_ornek" {

```

Daha sonra Name\_Tag'i Terr olan EC2-Instance'ımızın ID'sini kopyalayalım.

Name ▾	Instance ID
Terr	i-03f7522ad7424c0a1

Instance ID'yi import etmek istediğimiz kaynağa ekleyelim

```
$ terraform import aws_instance.import_ornek i-03f7522ad7424c0a1
```


İşlem tamamlandığında import ettiğimiz kaynağın bilgileri state dosyasının içinde olacaktır.

Hazır instance'ın IP adresleri:

Public IPv4 address

 3.86.35.124 [open address](#) 

Private IPv4 addresses

 172.31.88.192

Import ettikten sonra state file içine eklenen IP adresler:

```

"private_dns": "ip-172-31-88-192.ec2.internal",
"private_ip": "172.31.88.192",
"public_dns": "ec2-3-86-35-124.compute-1.amazonaws.com",
"public_ip": "3.86.35.124",
"root_block_device": [
  {

```

Bundan sonra benim yapmam gereken `terraform show` diyerek ihtiyacım olan argümanları kodumun içine eklemek.

```
provider "aws" {  
  region = "us-east-1"  
}  
  
resource "aws_instance" "import_ornek" {  
  ami          = "ami-0ed9277fb7eb570c9"  
  key_name     = "firstkey"  
  instance_type = "t2.micro"  
}
```