ΘΕΩΡΙΑ, ΑΣΚΗΣΕΙΣ ΚΑΙ ΕΡΩΤΗΣΕΙΣ ΕΞΕΤΑΣΗΣ

Terraform

από το πρώτο apply
μέχρι την πιστοποίηση

Γράφεις την υποδομή που θέλεις. Διαβάζεις το plan. Μετά κάνεις apply.



28 **κεφάλαια, 172 ασκήσεις, 223 ερωτήσεις**

ΚΑΘΕ ΕΝΤΟΛΗ ΕΚΤΕΛΕΣΤΗΚΕ ΜΕ TERRAFORM 1.16.4

ΠΡΙΝ ΑΠΟ ΤΟ ΠΡΩΤΟ APPLY

Περιγράφεις την υποδομή. Το Terraform βρίσκει τη διαφορά.

Με το Terraform δεν γράφεις βήματα του τύπου «φτιάξε αυτό, μετά άλλαξε εκείνο». Γράφεις πώς πρέπει να είναι η υποδομή. Το Terraform συγκρίνει την περιγραφή με αυτό που υπάρχει, σου δείχνει τη διαφορά σε ένα plan και, αν συμφωνήσεις, κάνει μόνο τις αλλαγές που λείπουν. Όλο το βιβλίο χτίζεται πάνω σε αυτόν τον κύκλο.

Ξεκινά χωρίς cloud. Στα πρώτα τρία μέρη τα resources είναι αρχεία στον δίσκο σου, τυχαίες τιμές και μικρά archives. Έτσι μαθαίνεις το workflow, το state, τη γλώσσα HCL και τα modules χωρίς λογαριασμό και χωρίς κόστος. Από το τέταρτο μέρος περνάς στο AWS. Φτιάχνεις buckets, δίκτυα και web servers, βάζεις το state σε S3 με locking, διορθώνεις drift, κάνεις import και αλλάζεις τη δομή του κώδικα χωρίς να χαθεί τίποτα.

Θεωρεί ότι ξέρεις να δουλεύεις σε terminal και ότι έχεις γράψει λίγο κώδικα. Δεν θεωρεί ότι έχεις γράψει Terraform. Κάθε έννοια εξηγείται εδώ, μέσα στο κεφάλαιο, πριν τη χρειαστείς σε άσκηση.

Πώς δουλεύεται

Κάθε εντολή του βιβλίου εκτελέστηκε κατά την παραγωγή του με Terraform 1.16.4 και η έξοδος κάτω από αυτήν είναι η πραγματική. Τα κεφάλαια του AWS εκτελέστηκαν στο Moto, έναν εξομοιωτή του AWS API που τρέχει τοπικά. Εσύ μπορείς να κάνεις το ίδιο ή να χρησιμοποιήσεις έναν sandbox λογαριασμό AWS. Τα IDs των resources θα είναι διαφορετικά στη δική σου εκτέλεση.

Κάθε κεφάλαιο κλείνει με ασκήσεις και ερωτήσεις στο ύφος της εξέτασης Terraform Associate 004. Κάτω από κάθε άσκηση υπάρχει το αναμενόμενο αποτέλεσμα, ώστε να ελέγχεις τη δουλειά σου χωρίς να κοιτάξεις τη λύση. Οι λύσεις και οι απαντήσεις με εξηγήσεις βρίσκονται στον χωριστό τόμο.



Δεν τρέχουμε ποτέ apply χωρίς να έχουμε διαβάσει το plan. Αυτός ο κανόνας ισχύει από την πρώτη σελίδα μέχρι την τελευταία.

Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

1 Infrastructure as Code και το πρώτο apply 5
2 Η γλώσσα HCL, fmt και validate 19
3 Το plan και οι αλλαγές 39
4 Το state 59
5 Providers, εκδόσεις και το lock file 82
6 Resources, references και dependencies 102
7 Input variables 126
8 Outputs και locals 149
9 Τύποι και δομές δεδομένων 173
10 Εκφράσεις και συναρτήσεις 192
11 count, for_each και dynamic blocks 216
12 Data sources 242
13 Validation, preconditions και check blocks 269
14 Lifecycle και αντικατάσταση 294
15 Το πρώτο module 321
16 Πηγές modules, εκδόσεις και Registry 346
17 Ο AWS provider 376
18 Δίκτυο και ένας web server 401
19 Backends, remote state και locking 426
20 Workspaces 453
21 Drift, refresh-only και εντολές state 482
22 Import 510
23 Refactoring με moved και removed 538
24 Secrets, sensitive, ephemeral και Vault 564
25 Troubleshooting και logging 592
26 HCP Terraform 621
27 Τελικό project 641
28 Προσομοίωση εξέτασης 664



ΜΕΡΟΣ Ι

Το βασικό workflow

Πριν από το cloud μαθαίνουμε τον κύκλο. Γράφεις τι θέλεις να υπάρχει, διαβάζεις τι θα αλλάξει και μόνο τότε αλλάζεις τον κόσμο. Τα resources αυτού του μέρους είναι αρχεία στον δίσκο σου, οπότε κάθε λάθος κοστίζει ένα `destroy` και τίποτα παραπάνω.



ΚΕΦΑΛΑΙΟ 1

Infrastructure as Code και το πρώτο apply

Το Terraform φτιάχνει υποδομή από αρχεία κειμένου. Σε αυτό το κεφάλαιο καταλαβαίνεις τι κερδίζεις από αυτό, εγκαθιστάς το Terraform και περνάς μία φορά από όλο τον κύκλο ζωής ενός resource, από τη δημιουργία μέχρι τη διαγραφή.

Από τα κλικ στον κώδικα

Φαντάσου ότι χρειάζεσαι έναν web server στο AWS. Ο πρώτος τρόπος είναι η κονσόλα. Ανοίγεις τη σελίδα του EC2, διαλέγεις image, μέγεθος, δίκτυο και κανόνες firewall, πατάς κουμπιά και ο server εμφανίζεται. Δουλεύει. Έναν μήνα μετά όμως κανείς δεν θυμάται ποια κουμπιά πατήθηκαν. Όταν χρειαστείς τον ίδιο server για δοκιμές, θα τον ξαναφτιάξεις από μνήμης και θα βγει λίγο διαφορετικός.

Ο δεύτερος τρόπος είναι ένα script που καλεί το API βήμα βήμα. Φτιάξε το δίκτυο, μετά το firewall, μετά τον server. Αυτό είναι **προστακτικό** (imperative) μοντέλο. Γράφεις τις ενέργειες. Το script δουλεύει την πρώτη φορά. Αν το τρέξεις δεύτερη φορά θα προσπαθήσει να φτιάξει δεύτερο δίκτυο, εκτός αν γράψεις κώδικα που ελέγχει τι υπάρχει ήδη. Αν αλλάξεις το μέγεθος του server θα χρειαστείς άλλο script για την αλλαγή.

Το **Infrastructure as Code** (IaC) σημαίνει ότι η υποδομή περιγράφεται σε αρχεία που διαβάζει μηχανή και που ζουν μαζί με τον υπόλοιπο κώδικα. Το Terraform ακολουθεί το **δηλωτικό** (declarative) μοντέλο. Δεν γράφεις ενέργειες. Γράφεις την κατάσταση που θέλεις, για παράδειγμα «ένας server τάδε μεγέθους σε αυτό το δίκτυο». Το Terraform κοιτάζει τι υπάρχει, βρίσκει τη διαφορά και κάνει μόνο όσα χρειάζονται για να την κλείσει. Αν τρέξεις τον ίδιο κώδικα δέκα φορές, το αποτέλεσμα είναι ένας server, όχι δέκα.

Τα πλεονεκτήματα φαίνονται μόλις η υποδομή μεγαλώσει ή αποκτήσει δεύτερο άνθρωπο.

- **Επαναληψιμότητα.** Ο ίδιος κώδικας φτιάχνει την ίδια υποδομή για development, staging και production.
- **Ιστορικό.** Τα αρχεία μπαίνουν σε Git. Ξέρεις ποιος άλλαξε τι, πότε και γιατί. Μπορείς επίσης να γυρίσεις σε προηγούμενη έκδοση.
- **Έλεγχος πριν από την αλλαγή.** Μια αλλαγή περνά από code review όπως κάθε άλλος κώδικας. Το Terraform δείχνει επιπλέον τι ακριβώς θα κάνει πριν το κάνει.
- **Τεκμηρίωση.** Ο κώδικας λέει τι υπάρχει. Δεν χρειάζεται χωριστό έγγραφο που ξεχνά κανείς να ενημερώσει.
- **Ταχύτητα και λιγότερα λάθη.** Ό,τι κάνει η μηχανή δεν το ξεχνά και δεν το κάνει μισό. Ένα περιβάλλον που χρειαζόταν μια μέρα κλικ στήνεται σε λεπτά.

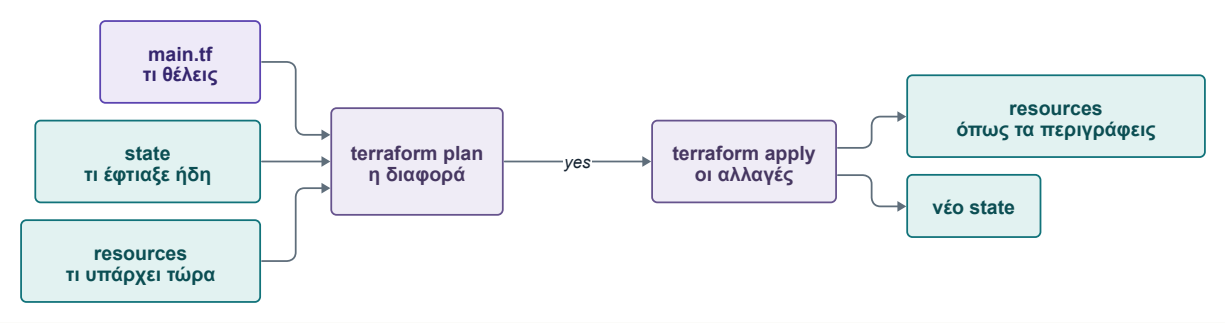
Υπάρχουν και άλλα εργαλεία IaC. Κάποια είναι δεμένα σε έναν πάροχο, όπως το CloudFormation του AWS. Άλλα, όπως το Ansible, είναι κυρίως εργαλεία configuration management που ρυθμίζουν λογισμικό μέσα σε servers που ήδη υπάρχουν. Το Terraform εστιάζει στο provisioning, δηλαδή στη δημιουργία και στη διαχείριση των ίδιων των resources. Μιλά με εκατοντάδες πλατφόρμες μέσω plugins που λέγονται providers. Τους γνωρίζεις αναλυτικά στο κεφάλαιο 5.

Πώς δουλεύει το Terraform

Το Terraform έχει τρία κομμάτια που θα συναντάς σε κάθε κεφάλαιο.

1. Η **configuration** είναι τα αρχεία `.tf` που γράφεις. Περιγράφουν τα resources που θέλεις να υπάρχουν.
2. Οι **providers** είναι plugins που ξέρουν να μιλούν με μια πλατφόρμα. Ο provider του AWS ξέρει το API του AWS. Ο provider `local`, που χρησιμοποιούμε σε αυτό το κεφάλαιο, ξέρει να φτιάχνει αρχεία στον δίσκο.
3. Το **state** είναι ένα αρχείο όπου το Terraform σημειώνει ποια πραγματικά αντικείμενα έφτιαξε και σε ποιο κομμάτι της configuration αντιστοιχεί το καθένα. Είναι η μνήμη του και το θέμα του κεφαλαίου 4.

Η δουλειά γίνεται σε έναν κύκλο με τρία βήματα. **Write**, γράφεις ή αλλάζεις την configuration. **Plan**, το Terraform συγκρίνει την configuration με το state και με τον πραγματικό κόσμο και σου δείχνει τι θα αλλάξει. **Apply**, αν συμφωνείς με το plan, το Terraform κάνει τις αλλαγές και ενημερώνει το state. Πριν από τον πρώτο κύκλο τρέχει μία φορά το `terraform init`, που ετοιμάζει τον φάκελο.



Σχήμα 1.1 · Ο κύκλος του Terraform. Το plan συγκρίνει όσα θέλεις με όσα θυμάται το state και με όσα υπάρχουν πραγματικά. Το apply κλείνει τη διαφορά και γράφει νέο state.



Εγκατάσταση

Το Terraform είναι ένα εκτελέσιμο αρχείο. Σε macOS το εγκαθιστάς από το επίσημο Homebrew tap της HashiCorp.

TERMINAL

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

Σε Debian και Ubuntu προσθέτεις το apt repository της HashiCorp και εγκαθιστάς το πακέτο `terraform`. Σε Windows υπάρχει πακέτο για `winget` και για `choco`. Σε κάθε σύστημα μπορείς επίσης να κατεβάσεις το zip από τη σελίδα releases της HashiCorp και να βάλεις το εκτελέσιμο σε φάκελο του `PATH`.

Η εντολή `terraform version` λέει ποια έκδοση έχεις και για ποια πλατφόρμα χτίστηκε.

TERMINAL

```
terraform version
```

ΕΞΟΔΟΣ

```
Terraform v1.16.4
on darwin_arm64
```

Το βιβλίο εκτελέστηκε με την 1.16.4. Οποιαδήποτε 1.x από την 1.12 και μετά δίνει την ίδια συμπεριφορά σε όσα θα δεις, με λίγες εξαιρέσεις που σημειώνονται όπου εμφανίζονται. Η δεύτερη γραμμή θα γράφει τη δική σου πλατφόρμα, για παράδειγμα `linux_amd64`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση Terraform Associate 004 βασίζεται στο Terraform 1.12. Ό,τι προστέθηκε αργότερα δεν εξετάζεται. Το βιβλίο το σημειώνει κάθε φορά που χρησιμοποιεί κάτι νεότερο.

Ο φάκελος εργασίας

Το Terraform δουλεύει πάντα σε έναν φάκελο. Διαβάζει όλα τα αρχεία `.tf` του φακέλου στον οποίο βρίσκεσαι και τα αντιμετωπίζει σαν μία configuration. Ο φάκελος αυτός λέγεται **working directory** ή **root module**.

Σε όλο το βιβλίο κάθε κεφάλαιο έχει δικό του φάκελο κάτω από το `~/tfbook`. Για αυτό το κεφάλαιο είναι το `~/tfbook/ch01`. Οι εντολές ξεκινούν από εκεί. Φτιάξε έναν φάκελο για το πρώτο παράδειγμα και μπες μέσα.

TERMINAL

```
mkdir hello
cd hello
pwd
```

ΕΞΟΔΟΣ

```
/home/you/tfbook/ch01/hello
```

Η πρώτη configuration

Θέλουμε ένα αρχείο κειμένου με όνομα `hello.txt` και συγκεκριμένο περιεχόμενο. Θα μπορούσες να το φτιάξεις με `echo`. Εδώ θα το περιγράψεις και θα το φτιάξει το Terraform, ακριβώς όπως αργότερα θα περιγράφεις buckets και servers. Γράψε στο `main.tf` τα εξής.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "hello" {
  filename = "hello.txt"
  content  = "Γεια σου, Terraform\n"
}
```

Το αρχείο έχει δύο **blocks**. Ένα block ξεκινά με μια λέξη που λέει τι είδους block είναι, μπορεί να έχει labels σε εισαγωγικά και κλείνει το περιεχόμενό του σε άγκιστρα. Το κεφάλαιο 2 εξηγεί τη σύνταξη αναλυτικά. Για τώρα αρκεί να διαβάζεις τα δύο blocks.

Το block `terraform` ρυθμίζει το ίδιο το Terraform. Μέσα του το `required_providers` λέει ότι η configuration χρειάζεται τον provider με τοπικό όνομα `local`. Το `source` λέει από πού θα κατέβει, εδώ από τον οργανισμό `hashicorp` στο Terraform Registry. Το `version` καρφώνει την έκδοση 2.9.1, ώστε να βλέπεις την ίδια έξοδο με το βιβλίο.

Το block `resource` περιγράφει ένα αντικείμενο που θέλουμε να υπάρχει. Έχει δύο labels. Το πρώτο, `local_file`, είναι ο **τύπος** του resource. Το πρόθεμα πριν από την πρώτη κάτω παύλα λέει σε ποιον provider ανήκει. Το δεύτερο, `hello`, είναι το **όνομα** που του δίνεις εσύ. Με αυτό θα το αναφέρεις στην

υπόλοιπη configuration. Τύπος και όνομα μαζί, `local_file.hello`, ταυτοποιούν το resource μέσα στη configuration.

Μέσα στο block υπάρχουν δύο **arguments**. Το `filename` λέει πού θα γραφτεί το αρχείο, σχετικά με τον φάκελο όπου τρέχεις το Terraform. Το `content` είναι το περιεχόμενο. Το `\n` στο τέλος είναι αλλαγή γραμμής.

terraform init

Ένας καινούργιος φάκελος πρέπει πρώτα να προετοιμαστεί. Το `terraform init` διαβάζει τη configuration, βρίσκει ποιους providers χρειάζεται, τους κατεβάζει και ρυθμίζει πού θα κρατιέται το state.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/local versions matching "2.9.1"...
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Διάβασε την έξοδο από πάνω προς τα κάτω. Η πρώτη ενότητα, `Initializing the backend`, αφορά το πού θα αποθηκεύεται το state. Δεν ρυθμίσαμε τίποτα, οπότε το Terraform θα το κρατά σε τοπικό αρχείο. Η δεύτερη ενότητα κατεβάζει τον provider. Βρήκε την έκδοση που ζητήσαμε, την εγκατέστησε και επιβεβαίωσε ότι είναι υπογεγραμμένη από τη HashiCorp.

Στο τέλος αναφέρει το **lock file**, το `.terraform.lock.hcl`. Εκεί καταγράφει ποια έκδοση κάθε provider διάλεξε, μαζί με checksums. Το αρχείο μπαίνει σε Git, ώστε όλοι όσοι δουλεύουν στο ίδιο project να παίρνουν την ίδια έκδοση. Το κεφάλαιο 5 το ανοίγει.

TERMINAL

```
ls -a
```

ΕΞΟΔΟΣ

```
.
..
.terraform
.terraform.lock.hcl
main.tf
```

Εκτός από το lock file εμφανίστηκε ο φάκελος `.terraform`. Εκεί μπήκε το εκτελέσιμο του provider. Δεν μπαίνει σε Git και μπορείς να τον σβήσεις οποτεδήποτε, γιατί ένα νέο `terraform init` τον ξαναφτιάχνει.

ΚΡΑΤΑ ΑΥΤΟ

Το `terraform init` τρέχει πρώτο σε κάθε νέο φάκελο και ξανά κάθε φορά που αλλάζεις providers, modules ή backend. Είναι ασφαλές να το τρέξεις όσες φορές θέλεις. Δεν αλλάζει τίποτα στην υποδομή σου.

terraform plan

Το `terraform plan` απαντά στην ερώτηση «τι θα γίνει αν κάνω apply τώρα;». Δεν αλλάζει τίποτα. Μόνο συγκρίνει και αναφέρει.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.hello will be created
  + resource "local_file" "hello" {
      + content              = <<-EOT
            Γεια σου, Terraform
        EOT
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "hello.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το Terraform λέει ότι θα κάνει μία ενέργεια, `create`, με σύμβολο `+`. Μετά δείχνει όλα τα attributes του `local_file.hello` όπως θα είναι μετά τη δημιουργία. Τα `content` και `filename` είναι όσα γράψαμε. Τα

`directory_permission` και `file_permission` είναι προεπιλογές του provider, που δεν τις ορίσαμε αλλά ισχύουν. Αρκετά attributes γράφουν `(known after apply)`. Η τιμή τους θα υπάρξει μόνο όταν φτιαχτεί το αρχείο, για παράδειγμα τα checksums του περιεχομένου και το `id`.

Η γραμμή `Plan: 1 to add, 0 to change, 0 to destroy.` είναι η περίληψη. Θα τη διαβάζεις σε κάθε plan, γιατί λέει με μια ματιά αν το plan κάνει αυτό που περιμένεις. Η σημείωση στο τέλος λέει ότι δεν αποθηκεύσαμε το plan. Το κεφάλαιο 3 δείχνει πώς αποθηκεύεται και γιατί αξίζει.

TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
```

Το `hello.txt` δεν υπάρχει ακόμη. Το plan δεν αλλάζει τίποτα.

terraform apply

Το `terraform apply` φτιάχνει πρώτα ένα νέο plan, σου το δείχνει και ρωτά αν θέλεις να συνεχίσεις. Δέχεται μόνο τη λέξη `yes`. Οτιδήποτε άλλο ακυρώνει.

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.hello will be created
  + resource "local_file" "hello" {
      + content              = <<-EOT
            Γεια σου, Terraform
        EOT
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "hello.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
```

```
  Enter a value: yes

local_file.hello: Creating...
local_file.hello: Creation complete after 0s [id=385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Μετά το `yes` το Terraform εκτέλεσε το plan. Η γραμμή `Creating...` λέει ότι ξεκίνησε η δημιουργία και η `Creation complete` ότι τελείωσε. Σε αγκύλες γράφει το `id` που πήρε το resource. Για τον provider `local` το `id` είναι checksum του περιεχομένου. Σε ένα S3 bucket θα ήταν το όνομα του bucket και σε έναν server του EC2 το instance ID. Η περίληψη στο τέλος επιβεβαιώνει ότι προστέθηκε ένα resource.

TERMINAL

```
cat hello.txt
ls
```

ΕΞΟΔΟΣ

```
Γεια σου, Terraform
hello.txt
main.tf
terraform.tfstate
```

Το αρχείο υπάρχει με το περιεχόμενο που περιγράψαμε. Δίπλα του εμφανίστηκε το `terraform.tfstate`. Εκεί το Terraform σημείωσε ότι το resource `local_file.hello` είναι το αρχείο `hello.txt` με αυτό το `id`. Χωρίς αυτή τη σημείωση την επόμενη φορά δεν θα ήξερε ότι το αρχείο το έφτιαξε το ίδιο.

Το δεύτερο plan

Τρέξε ξανά το plan χωρίς να αλλάξεις τίποτα.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.hello: Refreshing state... [id=385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το Terraform διάβασε το state, έλεγξε το πραγματικό αρχείο και βρήκε ότι συμφωνεί με τη configuration. Δεν έχει τίποτα να κάνει. Εδώ φαίνεται η διαφορά από ένα script. Ένα script θα έφτιαχνε το αρχείο ξανά. Το Terraform περιγράφει κατάσταση και η κατάσταση ήδη ισχύει. Η γραμμή `Refreshing state...` στην αρχή είναι ο έλεγχος του πραγματικού αρχείου. Το κεφάλαιο 4 εξηγεί τι ακριβώς κάνει.

Η ιδιότητα να δίνει το ίδιο αποτέλεσμα όσες φορές κι αν εκτελεστεί λέγεται **idempotence**. Είναι το πρώτο που κερδίζεις από το δηλωτικό μοντέλο.

Μια αλλαγή

Άλλαξε το περιεχόμενο. Το Terraform δεν χρειάζεται να μάθει τι άλλαξες. Θα το βρει μόνο του συγκρίνοντας.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "hello" {
  filename = "hello.txt"
  content  = "Γεια σου ξανά, Terraform\n"
}
```

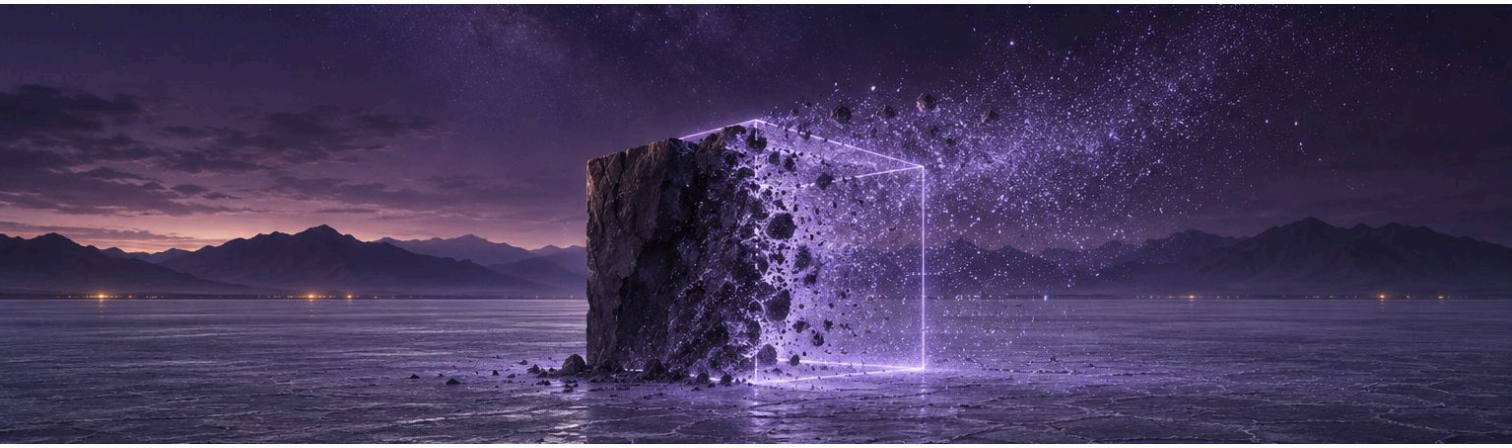
TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
local_file.hello: Refreshing state... [id=385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.hello must be replaced
-/+ resource "local_file" "hello" {
      ~ content              = <<-EOT # forces replacement
          - Γεια σου, Terraform
          + Γεια σου ξανά, Terraform
        EOT
      …
      ~ id                   = "385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

local_file.hello: Destroying... [id=385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07]
local_file.hello: Destruction complete after 0s
local_file.hello: Creating...
local_file.hello: Creation complete after 0s [id=25e4f0ef93649ad90afca9a4b9490fa1d66e8bde]

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Αυτή τη φορά το σύμβολο είναι `-/+` και η περίληψη λέει `1 to add` και `1 to destroy`. Το Terraform θα σβήσει το παλιό αρχείο και θα φτιάξει νέο. Δίπλα στο `content` γράφει `# forces replacement`. Ο provider



`local` δεν ξέρει να αλλάζει το περιεχόμενο ενός υπάρχοντος αρχείου, οπότε κάθε αλλαγή στο `content` σημαίνει αντικατάσταση. Άλλα resources αλλάζουν επί τόπου, με σύμβολο `~`. Το κεφάλαιο 3 εξηγεί πώς καταλαβαίνεις από πριν ποιο από τα δύο θα γίνει, γιατί σε έναν server η διαφορά είναι τεράστια.

TERMINAL

```
cat hello.txt
```

ΕΞΟΔΟΣ

```
Γεια σου ξανά, Terraform
```

terraform destroy

Το `terraform destroy` σβήνει όσα διαχειρίζεται η configuration. Δείχνει κι αυτό plan και ζητά `yes`.

TERMINAL

```
terraform destroy
```

ΕΞΟΔΟΣ

```
local_file.hello: Refreshing state... [id=25e4f0ef93649ad90afca9a4b9490fa1d66e8bde]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # local_file.hello will be destroyed
  - resource "local_file" "hello" {
      - content              = <<-EOT
            Γεια σου ξανά, Terraform
        EOT -> null
      …
      - directory_permission = "0777" -> null
      - file_permission      = "0777" -> null
```

```
      - filename             = "hello.txt" -> null
      - id                   = "25e4f0ef93649ad90afca9a4b9490fa1d66e8bde" -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

local_file.hello: Destroying... [id=25e4f0ef93649ad90afca9a4b9490fa1d66e8bde]
local_file.hello: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

Το σύμβολο είναι – και η περίληψη `1 to destroy`. Το destroy σβήνει μόνο όσα είναι γραμμένα στο state. Ένα αρχείο που έφτιαξες με το χέρι στον ίδιο φάκελο δεν το αγγίζει, γιατί το Terraform δεν ξέρει καν ότι υπάρχει.

TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
terraform.tfstate.backup
```

Το `hello.txt` έφυγε. Η configuration, το lock file και το state έμειναν. Το state είναι πλέον άδειο από resources. Δίπλα του εμφανίστηκε το `terraform.tfstate.backup`, ένα αντίγραφο του προηγούμενου state. Το κεφάλαιο 4 εξηγεί και τα δύο αρχεία. Αν ξανατρέξεις `terraform apply`, το αρχείο θα ξαναφτιαχτεί ίδιο. Αυτό είναι το νόημα του IaC. Η υποδομή μπορεί να εξαφανιστεί και να ξαναγεννηθεί από τον κώδικα.

ΠΑΓΙΔΑ

Μη σβήνεις το `terraform.tfstate` για να «ξεκινήσεις καθαρά». Αν το σβήσεις ενώ υπάρχουν resources, το Terraform ξεχνά ότι τα έφτιαξε. Το επόμενο apply θα προσπαθήσει να τα ξαναφτιάξει και το destroy δεν θα τα βρει. Ο σωστός τρόπος να καθαρίσεις είναι το `terraform destroy`.

Ο κύκλος σε μία σελίδα

Εντολή	Τι κάνει	Αλλάζει υποδομή
`terraform init`	Ετοιμάζει τον φάκελο, κατεβάζει providers	Όχι
`terraform plan`	Δείχνει τι θα αλλάξει	Όχι
`terraform apply`	Δείχνει plan, ζητά `yes`, εκτελεί	Ναι

Εντολή	Τι κάνει	Αλλάζει υποδομή
`terraform destroy`	Δείχνει plan διαγραφής, ζητά `yes`, σβήνει	Ναι

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά συχνά τη σειρά του workflow και τι κάνει κάθε εντολή. Write, plan, apply είναι ο βασικός κύκλος και το `init` προηγείται. Το `plan` δεν αλλάζει ποτέ πραγματική υποδομή. Το `destroy` είναι ισοδύναμο με ένα `apply` που θέλει να σβήσει τα πάντα. Το κεφάλαιο 3 δείχνει ότι γράφεται και `terraform apply -destroy`.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch01`. Γράψε το `main.tf` από την αρχή, τρέξε `terraform init` και μετά τις εντολές που ζητά η εκφώνηση.

ΑΣΚΗΣΗ 1.1 Δύο αρχεία

Γράψε configuration με δύο resources `local_file`. Το πρώτο, με όνομα `morning`, φτιάχνει το `morning.txt` με περιεχόμενο `Καλημέρα\n`. Το δεύτερο, με όνομα `night`, φτιάχνει το `night.txt` με περιεχόμενο `Καληνύχτα\n`. Κάνε apply. Η περίληψη στο τέλος πρέπει να είναι η παρακάτω.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

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

ΑΣΚΗΣΗ 1.2 Αρχείο σε υποφάκελο

Γράψε configuration που φτιάχνει το αρχείο `notes/today.txt` με περιεχόμενο `Πρώτη μέρα με Terraform\n`. Ο φάκελος `notes` δεν υπάρχει. Κάνε apply και μετά δείξε το περιεχόμενο με `cat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Πρώτη μέρα με Terraform
```

ΑΣΚΗΣΗ 1.3 Μόνο plan

Γράψε configuration με τρία resources `local_file`, τα `a.txt`, `b.txt` και `c.txt`, με όποιο περιεχόμενο θέλεις. Πριν τρέξεις οτιδήποτε, γράψε σε χαρτί τι πιστεύεις ότι θα λέει η περίληψη του plan. Τρέξε μόνο `terraform plan` και έλεγξε ότι δεν δημιουργήθηκε κανένα αρχείο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Plan: 3 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 1.4 Idempotence

Ξεκίνα από το `main.tf` της πρώτης άσκησης με τα δύο αρχεία. Κάνε apply δύο φορές στη σειρά χωρίς καμία αλλαγή ανάμεσα. Τι λέει το δεύτερο apply;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
local_file.night: Refreshing state... [id=1f630aef20dd5e60a76b660b707a914bdd029522]
local_file.morning: Refreshing state... [id=54a00210345bd8a3b7d8086cbb1f2887e061693c]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.

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

ΑΣΚΗΣΗ 1.5 Από την αρχή μέχρι το τέλος

Γράψε configuration για το αρχείο `cycle.txt` με περιεχόμενο `ένας κύκλος\n`. Κάνε apply. Μετά κάνε destroy και έλεγξε με `ls` ότι το αρχείο έφυγε ενώ το state έμεινε. Το αναμενόμενο αποτέλεσμα είναι η έξοδος του `ls`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
main.tf
terraform.tfstate
terraform.tfstate.backup
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 1.1

Μια ομάδα στήνει servers από την κονσόλα του cloud και κρατά σημειώσεις σε wiki. Ποιο πρόβλημα λύνει πρώτα η μετάβαση σε Infrastructure as Code;

a. Μειώνει το κόστος κάθε server.
b. Κάνει τη δημιουργία της υποδομής επαναλήψιμη και καταγεγραμμένη σε κώδικα.
c. Καταργεί την ανάγκη για πρόσβαση στο API του cloud.
d. Εγγυάται ότι η υποδομή δεν θα έχει ποτέ σφάλματα.

ΕΡΩΤΗΣΗ 1.2

Ποια εντολή πρέπει να τρέξει πρώτη σε έναν φάκελο με νέα configuration;

a. `terraform plan`
b. `terraform validate`
c. `terraform init`
d. `terraform apply`

ΕΡΩΤΗΣΗ 1.3

Τρέχεις `terraform plan` σε μια configuration που περιγράφει τρία resources που δεν υπάρχουν. Τι συμβαίνει στην υποδομή;

a. Δημιουργούνται τα τρία resources χωρίς ερώτηση.
b. Τίποτα. Το plan μόνο δείχνει ότι θα προστεθούν τρία resources.
c. Δημιουργείται το πρώτο resource ως δοκιμή.
d. Γράφονται τα τρία resources στο state ως εκκρεμή.

ΕΡΩΤΗΣΗ 1.4

Τρέχεις `terraform apply` δεύτερη φορά χωρίς να έχεις αλλάξει τίποτα στη configuration ή στην υποδομή. Τι κάνει το Terraform;

a. Αναφέρει ότι δεν υπάρχουν αλλαγές και δεν κάνει τίποτα.
b. Ξαναφτιάχνει όλα τα resources με τις ίδιες ρυθμίσεις.
c. Αποτυγχάνει επειδή τα resources υπάρχουν ήδη.
d. Σβήνει τα resources και τα ξαναφτιάχνει.

ΕΡΩΤΗΣΗ 1.5

Σωστό ή λάθος; Το `terraform destroy` σβήνει μόνο τα resources που καταγράφονται στο state του τρέχοντος φακέλου.

a. Σωστό
b. Λάθος



ΚΕΦΑΛΑΙΟ 2

Η γλώσσα HCL, fmt και validate

Στο κεφάλαιο 1 έγραψες δύο blocks ακολουθώντας ένα πρότυπο. Εδώ μαθαίνεις τους κανόνες της γλώσσας, πώς μοιράζεις μια configuration σε πολλά αρχεία και δύο εντολές που δεν αγγίζουν ποτέ την υποδομή. Το `terraform fmt` τακτοποιεί τη μορφή του κώδικα. Το `terraform validate` βρίσκει λάθη πριν από το plan.

Blocks, labels και arguments

Οι configurations γράφονται σε **HCL**, τη HashiCorp Configuration Language. Είναι δηλωτική γλώσσα. Δεν γράφεις βήματα, περιγράφεις αντικείμενα. Έχει δύο δομικά στοιχεία, τα **blocks** και τα **arguments**. Φτιάξε έναν φάκελο για τα παραδείγματα σύνταξης.

TERMINAL

```
mkdir syntax
cd syntax
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "readme" {
  filename        = "README.txt"
  content         = "Παραδείγματα σύνταξης HCL\n"
  file_permission = "0644"
}
```

Ένα **block** έχει τρία μέρη. Πρώτα τον **τύπο** του, μια λέξη χωρίς εισαγωγικά, εδώ `terraform` και `resource`. Μετά μηδέν ή περισσότερα **labels** σε εισαγωγικά. Τέλος το **σώμα** σε άγκιστρα, με arguments και άλλα blocks. Πόσα labels θέλει ένα block το ορίζει ο τύπος του. Το `resource` θέλει δύο, τον τύπο του resource και το όνομά του. Αν ξεχάσεις το όνομα, το Terraform απαντά `All resource blocks must have 2 labels (type, name).` Το `terraform` δεν θέλει κανένα.

Τύπος block	Labels	Κεφάλαιο
`terraform`	0	1
`resource`	2	1
`provider`	1	5
`variable`	1	7
`output`	1	8
`locals`	0	8
`data`	2	12
`module`	1	15

Ένα **argument** έχει τη μορφή `όνομα = τιμή`. Αριστερά βρίσκεται ένα όνομα και δεξιά μια **έκφραση** (expression), δηλαδή οτιδήποτε δίνει τιμή. Η πιο απλή έκφραση είναι μια σταθερή τιμή, όπως το `"README.txt"`. Ποια arguments δέχεται ένα resource το ορίζει ο provider του. Η σειρά τους μέσα στο block δεν παίζει ρόλο.

Το `required_providers` είναι block μέσα σε block, δηλαδή **nested block**. Μέσα του όμως το `local = { ... }` είναι argument. Η τιμή του είναι ζεύγη κλειδιού και τιμής σε άγκιστρα, ένα object, που εξηγείται στο κεφάλαιο 9. Η διαφορά φαίνεται στο `=`. Το argument το έχει, το block όχι.

> **ΠΑΓΙΔΑ**
>
> Αν γράψεις `required_providers = {`, το Terraform ρωτά αν ήθελες block. Αν γράψεις `local {` χωρίς `=`, απαντά `Blocks are not allowed here.`

Τα ονόματα χωρίς εισαγωγικά, όπως οι τύποι των blocks και τα ονόματα των arguments, λέγονται **identifiers**. Τους ίδιους κανόνες ακολουθεί και το όνομα ενός resource. Ξεκινούν με γράμμα ή κάτω

παύλα και περιέχουν μόνο γράμματα, ψηφία, κάτω παύλες και παύλες. Το `readme_2` είναι έγκυρο όνομα. Το `2readme` απορρίπτεται με `Invalid resource name`.

Σχόλια και απλές τιμές

Η HCL δέχεται τρία είδη σχολίων. Το `#` είναι η μορφή που προτείνει η HashiCorp.

HCL

```
# Σχόλιο μίας γραμμής, η συνηθισμένη μορφή.
// Κι αυτό σχόλιο μίας γραμμής.
/*
  Σχόλιο σε πολλές γραμμές. Χρήσιμο για να
  απενεργοποιήσεις προσωρινά ένα ολόκληρο block.
*/
```

Οι απλές τιμές είναι τριών ειδών. Το **string** γράφεται σε διπλά εισαγωγικά, όπως το `"README.txt"`. Το **number** γράφεται χωρίς εισαγωγικά, όπως το `42` ή το `3.14`. Το **bool** είναι το `true` ή το `false`, επίσης χωρίς εισαγωγικά.

Μέσα σε string η ανάποδη κάθετος ξεκινά ένα **escape sequence**, έναν συνδυασμό που σημαίνει έναν χαρακτήρα που δεν γράφεται εύκολα αλλιώς.

Escape	Σημαίνει
`\n`	αλλαγή γραμμής
`\t`	tab
`\"`	διπλό εισαγωγικό μέσα στο string
`\\`	μία ανάποδη κάθετος
`\u03a9`	ο χαρακτήρας Unicode με αυτόν τον κωδικό, εδώ το Ω

Ένα string σε εισαγωγικά δεν σπάει σε δύο γραμμές του κώδικα. Το Terraform το απορρίπτει με `Invalid multi-line string`. Πρόσθεσε στο τέλος του `main.tf` τα παρακάτω.

main.tf

```
# Escape sequences μέσα σε string
resource "local_file" "quote" {
  filename = "quote.txt"
  content  = "Είπε \"καλημέρα\" και έφυγε.\nΦάκελος C:\\temp\n\u03a9 σημαίνει ωμέγα.\n"
}

# Αριθμός και bool σε arguments που περιμένουν string
resource "local_file" "answer" {
  filename = "answer.txt"
  content  = 42 // χωρίς εισαγωγικά
}

resource "local_file" "flag" {
  filename = "flag.txt"
  content  = true
}

/*
```

```
resource "local_file" "draft" {
  filename = "draft.txt"
  content  = "Δεν θα φτιαχτεί\n"
}
*/
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
  # local_file.answer will be created
…
      + content              = "42"
…
  # local_file.flag will be created
…
      + content              = "true"
…
  # local_file.quote will be created
…
      + content              = <<-EOT
…
  # local_file.readme will be created
…
      + content              = <<-EOT
…
Plan: 4 to add, 0 to change, 0 to destroy.
…
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

Το plan έχει τέσσερα resources, γιατί το `draft` βρίσκεται μέσα σε σχόλιο. Στο `answer` έγραψες τον αριθμό `42`, αλλά το plan δείχνει `"42"` σε εισαγωγικά. Το `content` δέχεται μόνο string, οπότε το Terraform μετέτρεψε τον αριθμό αυτόματα. Το ίδιο έγινε με το `true`. Οι μετατροπές τύπων είναι θέμα του κεφαλαίου 9. Τα strings με αλλαγές γραμμής το plan τα δείχνει ως `<<-EOT`, όπως είδες και στο κεφάλαιο 1.

TERMINAL

```
cat quote.txt
cat answer.txt flag.txt
```

ΕΞΟΔΟΣ

```
Είπε "καλημέρα" και έφυγε.
Φάκελος C:\temp
Ω σημαίνει ωμέγα.
42true
```

Κάθε escape έγινε ο χαρακτήρας του. Τα `42` και `true` γράφτηκαν χωρίς αλλαγή γραμμής, γιατί δεν τη ζητήσαμε, οπότε το `cat` τα τύπωσε κολλητά.

Heredoc και interpolation

Για κείμενο πολλών γραμμών υπάρχει το **heredoc**. Ξεκινά με `<<` και μια λέξη της επιλογής σου, συνήθως `EOT`. Τελειώνει με μια γραμμή που έχει μόνο την ίδια λέξη. Το `<<EOT` κρατά το κείμενο ακριβώς όπως είναι. Το `<<-EOT` βρίσκει τη μικρότερη εσοχή των γραμμών και την αφαιρεί από όλες. Έτσι στοιχίζεις το κείμενο με τον κώδικα χωρίς να μπουν κενά στο αρχείο.

Ένα string ή ένα heredoc μπορεί επίσης να περιέχει **interpolation**, μια έκφραση ανάμεσα σε `${` και `}`. Το Terraform υπολογίζει την έκφραση και βάζει το αποτέλεσμα μέσα στο κείμενο. Για τους χαρακτήρες `${` αυτούσιους γράφεις `$${`.

main.tf

```
resource "local_file" "poem" {
  filename = "poem.txt"
  content  = <<-EOT
    Πρώτη γραμμή χωρίς εσοχή.
      Δεύτερη με δύο κενά.
    Τρίτη χωρίς εσοχή.
  EOT
}

resource "local_file" "banner" {
  filename = "banner.txt"
  content  = <<EOT
    Τέσσερα κενά μπροστά.
Καμία εσοχή.
EOT
}

resource "local_file" "sum" {
  filename = "sum.txt"
  content  = "Δύο και δύο κάνουν ${2 + 2}.\nΤο $${2 + 2} μένει όπως είναι.\n"
}
```

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

TERMINAL

```
cat poem.txt banner.txt sum.txt
```



ΕΞΟΔΟΣ

```
Πρώτη γραμμή χωρίς εσοχή.
  Δεύτερη με δύο κενά.
Τρίτη χωρίς εσοχή.
    Τέσσερα κενά μπροστά.
Καμία εσοχή.
Δύο και δύο κάνουν 4.
Το ${2 + 2} μένει όπως είναι.
```

Στο `poem` η μικρότερη εσοχή ήταν τέσσερα κενά. Το `<<-EOT` τα αφαίρεσε και η δεύτερη γραμμή, που είχε έξι, κράτησε δύο. Στο `banner` το `<<EOT` δεν αφαίρεσε τίποτα. Στο `sum` το `${2 + 2}` έγινε `4` ενώ το `$${2 + 2}` έμεινε κείμενο. Με τον ίδιο τρόπο το `%{` ξεκινά οδηγίες όπως `if` και `for` και γράφεται αυτούσιο ως `%%{`. Το κεφάλαιο 10 εξηγεί τις εκφράσεις. Μέχρι τότε μέσα στο `${}` θα βλέπεις κυρίως αναφορές σε άλλα resources.

Πολλά αρχεία, μία configuration

Το Terraform διαβάζει κάθε αρχείο `.tf` του φακέλου εργασίας και τα ενώνει σε μία configuration. Το `main.tf` είναι σύμβαση, όχι κανόνας. Συνήθως το block `terraform` μπαίνει σε δικό του αρχείο, που σε αυτό το βιβλίο λέγεται `versions.tf`. Τα resources μπαίνουν στο `main.tf` και σε όσα ακόμη αρχεία βοηθούν την ανάγνωση.

TERMINAL

```
cd ..
mkdir split
cd split
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

main.tf

```
resource "local_file" "readme" {
  filename = "README.txt"
  content  = "Δες και το ${local_file.notes.filename}\n"
}
```

notes.tf

```
resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Σημειώσεις\n"
}
```

Το `local_file.notes.filename` στο `main.tf` είναι **αναφορά** (reference) στο argument `filename` ενός resource που ορίζεται σε άλλο αρχείο. Δεν δηλώνεις πουθενά ότι το ένα αρχείο χρησιμοποιεί το άλλο. Οι αναφορές εξηγούνται στο κεφάλαιο 6.

Το Terraform διαβάζει επίσης αρχεία `.tf.json`, με την ίδια δομή σε JSON. Τα γράφουν συνήθως προγράμματα, όχι άνθρωποι. Πρόσθεσε ένα τέτοιο αρχείο και ένα `.tf` σε υποφάκελο.

extra.tf.json

```
{
  "resource": {
    "local_file": {
      "from_json": {
        "filename": "from_json.txt",
        "content": "Γραμμένο σε JSON\n"
      }
    }
  }
}
```

drafts/old.tf

```
resource "local_file" "old" {
  filename = "old.txt"
  content  = "Παλιό πρόχειρο\n"
}
```

TERMINAL

```
find . -type f | sort
```

ΕΞΟΔΟΣ

```
./drafts/old.tf
./extra.tf.json
./main.tf
./notes.tf
./versions.tf
```

TERMINAL

```
terraform init
```

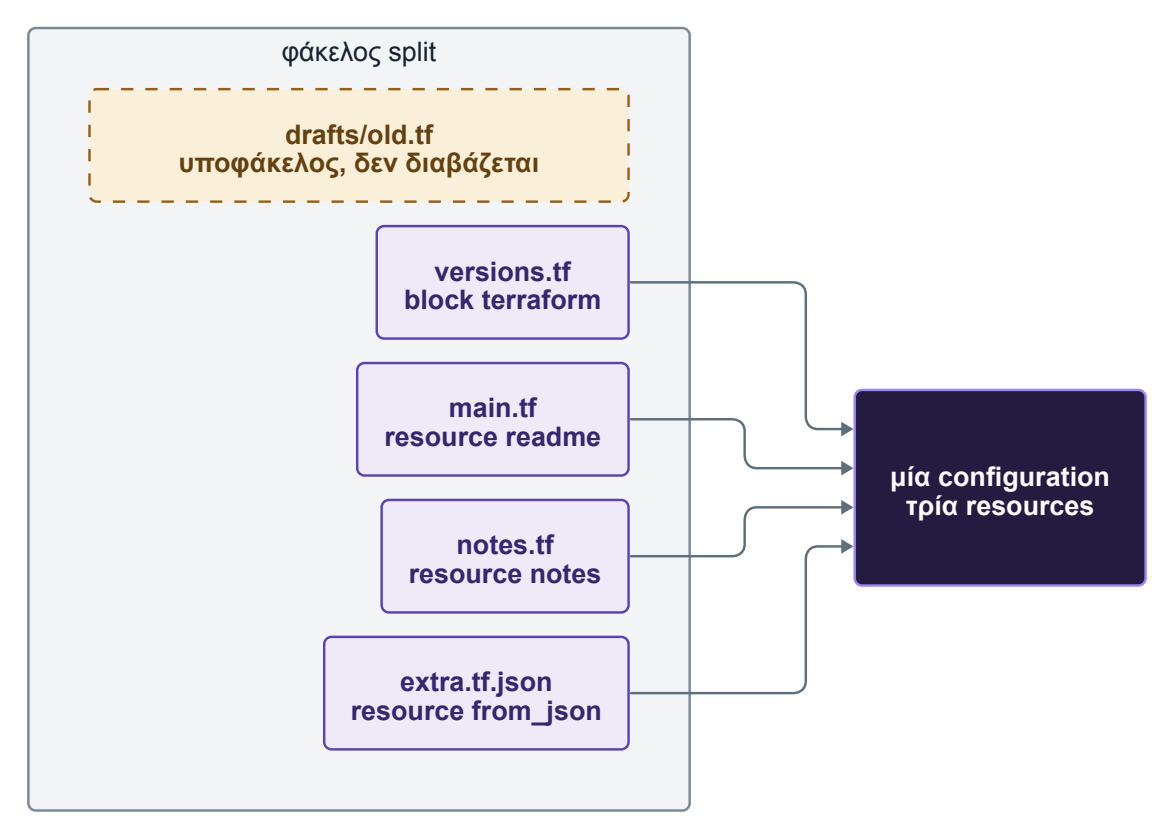
TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.from_json will be created
…
  # local_file.notes will be created
…
  # local_file.readme will be created
…
Plan: 3 to add, 0 to change, 0 to destroy.
…
```

Το plan έχει τρία resources από τρία αρχεία. Το `versions.tf` δεν έχει resources, αλλά δηλώνει τον provider. Το `local_file.old` λείπει, γιατί το Terraform δεν διαβάζει υποφακέλους. Ένας υποφάκελος με αρχεία `.tf` διαβάζεται μόνο όταν τον καλέσεις ως module, το θέμα του κεφαλαίου 15.



Σχήμα 2.1 · Τα αρχεία .tf και .tf.json του φακέλου ενώνονται σε μία configuration. Ο υποφάκελος drafts δεν διαβάζεται.

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
local_file.from_json: Creation complete after 0s [id=233c323b41055d054005d0cbf0c0291243d6b4e5]
local_file.notes: Creation complete after 0s [id=e0822e53b351b478ee1063027505072fcc51a072]
…
local_file.readme: Creation complete after 0s [id=82679cbe91f7edce261bbff26abb526134555c91]
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

TERMINAL

```
cat README.txt
```

ΕΞΟΔΟΣ

```
Δες και το notes.txt
```

Το `readme` φτιάχτηκε μετά το `notes`, γιατί χρειάζεται το `filename` του. Το Terraform διάβασε το `main.tf` πριν από το `notes.tf`, αφού διαβάζει τα αρχεία αλφαβητικά. Αυτό όμως δεν μετράει. Ενώνει πρώτα όλα τα αρχεία και μετά αποφασίζει τη σειρά των ενεργειών από τις αναφορές.

ΚΡΑΤΑ ΑΥΤΟ

Όλα τα `.tf` και `.tf.json` του φακέλου εργασίας είναι μία configuration. Η σειρά των αρχείων και των blocks δεν μετράει. Τα ονόματα των resources πρέπει να είναι μοναδικά σε όλο τον φάκελο.

ΠΑΓΙΔΑ

Αρχεία με όνομα `override.tf` ή που τελειώνουν σε `_override.tf` δεν ενώνονται κανονικά. Αλλάζουν arguments σε blocks των άλλων αρχείων. Μη δίνεις τέτοια ονόματα κατά λάθος.

terraform fmt

Το `terraform fmt` ξαναγράφει τα αρχεία στην κανονική μορφή της HCL. Όταν όλοι γράφουν με τους ίδιους κανόνες, ένα diff στο Git δείχνει μόνο πραγματικές αλλαγές. Φτιάξε ένα αρχείο γραμμένο βιαστικά.

TERMINAL

```
cd ..
mkdir messy
cd messy
```

main.tf

```
// Λίστα για το σούπερ μάρκετ
resource "local_file"   "shopping" {
filename="shopping.txt"
    content = "ψωμί\nγάλα\n"
      file_permission   =    "0644"
}
resource "local_file" "todo"{
  filename = "todo.txt"  # τι μένει
  content = <<-EOT
    πλύσιμο
    σιδέρωμα
  EOT
}
```

Το fmt δεν χρειάζεται `init`, γιατί δουλεύει μόνο με το κείμενο. Πρώτα έλεγξε χωρίς να αλλάξεις τίποτα.

TERMINAL

```
terraform fmt -check
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
main.tf
exit code 3
```

Με το `-check` το fmt δεν γράφει τίποτα. Τυπώνει τα αρχεία που θα άλλαζε και τελειώνει με exit code 3 αντί για 0, οπότε ένα CI pipeline αποτυγχάνει. Το `-diff` δείχνει τι θα αλλάξει, με `-` τις τωρινές γραμμές και με `+` τις νέες.

TERMINAL

```
terraform fmt -check -diff
```

ΕΞΟΔΟΣ

```
main.tf
--- old/main.tf
+++ new/main.tf
@@ -1,12 +1,12 @@
 // Λίστα για το σούπερ μάρκετ
-resource "local_file"   "shopping" {
-filename="shopping.txt"
-    content = "ψωμί\nγάλα\n"
-      file_permission   =    "0644"
+resource "local_file" "shopping" {
+  filename        = "shopping.txt"
+  content         = "ψωμί\nγάλα\n"
+  file_permission = "0644"
 }
-resource "local_file" "todo"{
-  filename = "todo.txt"  # τι μένει
-  content = <<-EOT
+resource "local_file" "todo" {
+  filename = "todo.txt" # τι μένει
+  content  = <<-EOT
     πλύσιμο
     σιδέρωμα
   EOT
```

TERMINAL

```
terraform fmt
cat main.tf
```

ΕΞΟΔΟΣ

```
main.tf
// Λίστα για το σούπερ μάρκετ
resource "local_file" "shopping" {
  filename        = "shopping.txt"
  content         = "ψωμί\nγάλα\n"
  file_permission = "0644"
}
resource "local_file" "todo" {
  filename = "todo.txt" # τι μένει
  content  = <<-EOT
    πλύσιμο
    σιδέρωμα
  EOT
}
```

Το fmt τύπωσε το αρχείο που ξανάγραψε. Έβαλε εσοχή δύο κενών, ένα κενό ανάμεσα στα labels και πριν από το `{` και στοίχισε τα `=` των διαδοχικών arguments. Πριν από το σχόλιο στο τέλος της γραμμής άφησε ένα κενό. Δεν άλλαξε το `//`, δεν πρόσθεσε κενή γραμμή ανάμεσα στα blocks και δεν άγγιξε το

περιεχόμενο του heredoc. Ονόματα, τιμές και σειρά έμειναν ίδια. Ένα argument με ορθογραφικό λάθος θα στοιχιστεί όμορφα και θα μείνει λάθος.

Χωρίς ορίσματα το fmt κοιτάζει μόνο τον τρέχοντα φάκελο. Πρόσθεσε ένα αρχείο σε υποφάκελο.

extra/more.tf

```
resource "local_file" "more" {
filename = "more.txt"
}
```

TERMINAL

```
terraform fmt -check
echo "exit code $?"
terraform fmt -recursive
```

ΕΞΟΔΟΣ

```
exit code 0
extra/more.tf
```

Το `-check` πέρασε με 0, γιατί δεν κοίταξε τον υποφάκελο. Το `-recursive` βρήκε και διόρθωσε το `extra/more.tf`. Σε repository με modules θέλεις σχεδόν πάντα το `-recursive`. Το fmt διορθώνει και τα αρχεία `.tfvars` του κεφαλαίου 7, αλλά όχι τα `.tf.json`. Σε αρχείο με συντακτικό λάθος δεν αλλάζει τίποτα. Τυπώνει το σφάλμα και τελειώνει με exit code 2.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform fmt` ξαναγράφει τα αρχεία στην κανονική μορφή και δεν χρειάζεται `init`. Το `-check` δεν αλλάζει αρχεία και επιστρέφει μη μηδενικό exit code όταν κάποιο θέλει αλλαγή, γι' αυτό μπαίνει στα CI pipelines. Το `-diff` δείχνει τις αλλαγές και το `-recursive` πιάνει τους υποφακέλους. Το fmt δεν ελέγχει αν η configuration είναι σωστή.

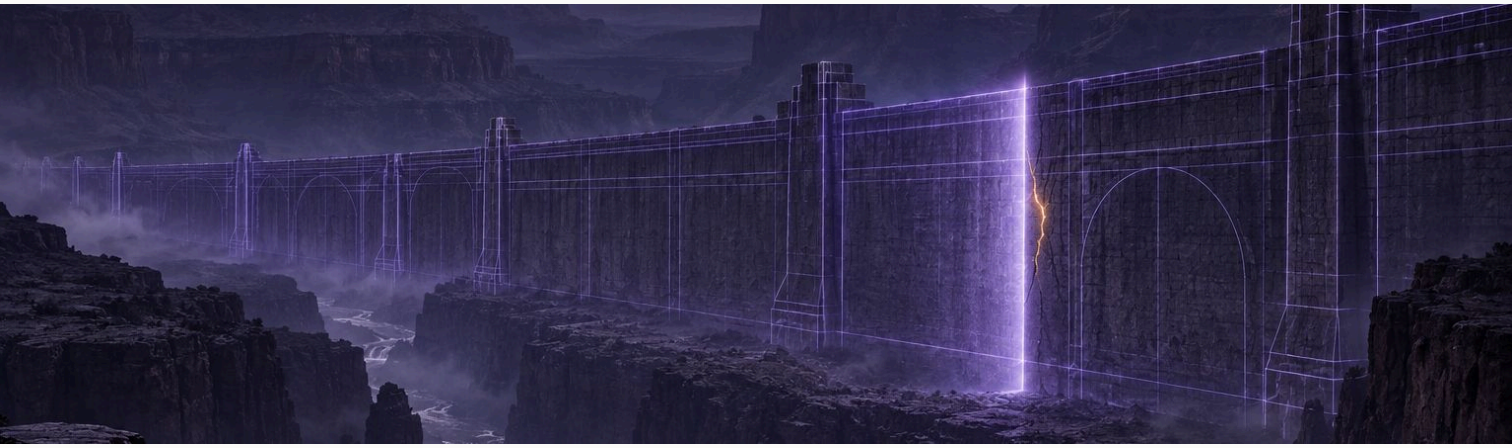
ΠΑΓΙΔΑ

Το `-diff` μόνο του γράφει κιόλας τις αλλαγές και επιστρέφει 0. Για να δεις χωρίς να αλλάξει τίποτα, γράψε `terraform fmt -check -diff`.

Συμβάσεις ονομάτων

Το fmt δεν αγγίζει ονόματα. Ο οδηγός στυλ της HashiCorp θέλει ουσιαστικά που λένε τι είναι το αντικείμενο, κάτω παύλες ανάμεσα στις λέξεις και καμία επανάληψη του τύπου στο όνομα, αφού η διεύθυνση του resource τον περιέχει ήδη. Στην πράξη όλοι γράφουν snake_case, δηλαδή πεζά λατινικά γράμματα, ψηφία και κάτω παύλες.

Αντί για	Γράψε
`local_file.readme_file`	`local_file.readme`
`local_file.ShoppingList`	`local_file.shopping_list`
`local_file.shopping-list`	`local_file.shopping_list`
`local_file.file1`	`local_file.shopping_list`



Διάλεξε καλό όνομα από την αρχή. Για το Terraform ένα resource με νέο όνομα είναι άλλο resource, οπότε η μετονομασία σημαίνει διαγραφή και νέα δημιουργία. Το κεφάλαιο 23 δείχνει πώς την αποφεύγεις.

Ο ίδιος οδηγός προτείνει και ονόματα αρχείων. Τα resources μπαίνουν στο `main.tf` και το block `terraform` στο `terraform.tf`. Πολλά projects, όπως και αυτό το βιβλίο, δίνουν στο δεύτερο το όνομα `versions.tf`. Αργότερα έρχονται τα `providers.tf`, `variables.tf` και `outputs.tf`. Το Terraform δεν κοιτάζει τα ονόματα των αρχείων, εκτός από τα override.

terraform validate

Το `terraform validate` ελέγχει αν η configuration είναι σωστή ως κείμενο. Ελέγχει σύνταξη και αναφορές. Ελέγχει επίσης τα ονόματα και τους τύπους των arguments απέναντι στο **schema** του provider, δηλαδή την περιγραφή κάθε resource που δίνει ο ίδιος ο provider. Δεν διαβάζει state και δεν καλεί κανένα API.

TERMINAL

```
cd ..
mkdir check
cd check
```

main.tf

```
resource "local_file" "readme" {
  filename = "README.txt"
  content  = "Διάβασέ με\n"
}
```

Αντίγραψε το `versions.tf` από τον φάκελο `split` και τρέξε validate χωρίς `init`.

TERMINAL

```
cp ../split/versions.tf .
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Missing required provider
│
│ This configuration requires provider registry.terraform.io/hashicorp/local,
│ but that provider isn't available. You may be able to install it
│ automatically by running:
│   terraform init
╵
```

Το schema έρχεται από τον provider, που δεν έχει κατέβει ακόμη. Μετά το `init` ο έλεγχος περνά.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

Τώρα χάλασε το `main.tf` με πέντε συνηθισμένα λάθη.

main.tf

```
resource "local_file" "readme" {
  contents = "Διάβασέ με\n"
}

resource "local_file" "notes" {
  filename = "notes.txt"
  content  = ["Σημειώσεις"]
}

resource "local_file" "index" {
  filename = "index.txt"
  content  = "Δες το ${local_file.todo.filename}\n"
}

resource "local_file" "secret" {
  filename        = "secret.txt"
  content         = "μυστικό\n"
  file_permission = "999"
}
```

TERMINAL

```
terraform validate
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
╷
│ Error: Missing required argument
│
│   on main.tf line 1, in resource "local_file" "readme":
│    1: resource "local_file" "readme" {
│
│ The argument "filename" is required, but no definition was found.
╵

╷
│ Error: Unsupported argument
│
│   on main.tf line 2, in resource "local_file" "readme":
│    2:   contents = "Διάβασέ με\n"
│
│ An argument named "contents" is not expected here. Did you mean "content"?
╵

╷
│ Error: Incorrect attribute value type
│
│   on main.tf line 7, in resource "local_file" "notes":
│    7:   content  = ["Σημειώσεις"]
│
│ Inappropriate value for attribute "content": string required, but have
│ tuple.
╵

╷
│ Error: Reference to undeclared resource
│
│   on main.tf line 12, in resource "local_file" "index":
│   12:   content  = "Δες το ${local_file.todo.filename}\n"
│
│ A managed resource "local_file" "todo" has not been declared in the root
│ module.
╵

╷
│ Error: Invalid File Permission String Value
│
│   with local_file.secret,
│   on main.tf line 18, in resource "local_file" "secret":
│   18:   file_permission = "999"
│
│ bad mode permission: string must be expressed in octal numeric notation:
│ 999
╵
exit code 1
```

Πώς διαβάζεται ένα diagnostic

Κάθε μήνυμα για πρόβλημα λέγεται **diagnostic** και έχει πάντα την ίδια μορφή. Πάρε το πρώτο.

1. Η γραμμή `Error: Missing required argument` είναι η περίληψη. Ένα `Warning` θα σήμαινε ότι η εντολή συνεχίζει.
2. Η γραμμή που ξεκινά με `on` λέει πού. Αρχείο `main.tf`, γραμμή 1, στο `resource "local_file" "readme"`.
3. Ακολουθεί η γραμμή του κώδικα. Στο terminal το κομμάτι που φταίει είναι υπογραμμισμένο.
4. Τέλος η εξήγηση. Το `filename` είναι υποχρεωτικό και λείπει.

Τα `╷`, `│` και `╵` απλώς πλαισιώνουν κάθε diagnostic. Το validate δεν σταματά στο πρώτο λάθος. Τύπωσε πέντε και τελείωσε με exit code 1.

- `Unsupported argument`. Το `contents` δεν υπάρχει και το Terraform προτείνει το `content`. Το `readme` έχει λοιπόν δύο λάθη, το `filename` που λείπει και το `contents`.

- `Incorrect attribute value type`. Το `content` θέλει string και πήρε λίστα σε αγκύλες, που το μήνυμα ονομάζει `tuple` (κεφάλαιο 9).
- `Reference to undeclared resource`. Δεν υπάρχει `local_file.todo`.
- `Invalid File Permission String Value`. Αυτό το έλεγξε ο provider, που δέχεται μόνο οκταδικά ψηφία. Η γραμμή `with local_file.secret,` λέει σε ποιο resource αφορά το μήνυμα. Εμφανίζεται συνήθως σε μηνύματα από provider.

Μερικά λάθη κρύβονται πίσω από άλλα. Σε resource με λάθος αναφορά ο provider δεν ελέγχει τις τιμές, οπότε ξανατρέχεις validate μετά από κάθε διόρθωση. Κάποια λάθη, όπως δύο resources με το ίδιο όνομα, σταματούν ακόμη και το `init`.

Τι δεν ελέγχει το validate

Το validate ελέγχει το κείμενο, όχι τον κόσμο. Δεν ξέρει αν έχεις δικαίωμα να φτιάξεις κάτι, αν υπάρχει ένα αντικείμενο που χρειάζεσαι ή αν το API θα δεχτεί μια τιμή. Φτιάξε έναν φάκελο όπου κανείς δεν μπορεί να γράψει.

TERMINAL

```
mkdir locked
chmod 555 locked
```

main.tf

```
resource "local_file" "report" {
  filename = "locked/report.txt"
  content  = "Αναφορά\n"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
  Enter a value: yes
…
local_file.report: Creating...
…
│ Error: Create local file error
…
│ +Original Error: open locked/report.txt: permission denied
…
```

Το validate πέρασε, το plan βγήκε κανονικά και ζήτησε `yes`. Το λάθος ήρθε από το λειτουργικό σύστημα τη στιγμή που ο provider έγραφε το αρχείο. Σε ένα cloud το αντίστοιχο είναι ένας χρήστης χωρίς δικαίωμα ή ένα API που απορρίπτει μια τιμή. Δώσε πίσω το δικαίωμα.

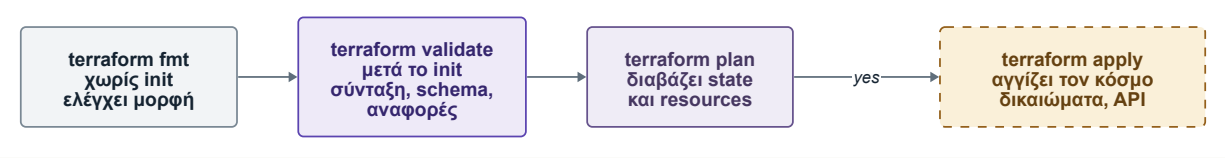
TERMINAL

```
chmod 755 locked
terraform apply
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Ο κώδικας δεν άλλαξε. Άλλαξε ο κόσμος και το apply πέτυχε. Αν δουλεύεις ως `root`, το `chmod 555` δεν σε σταματά και το πρώτο apply πετυχαίνει.



Σχήμα 2.2 · Τι ελέγχει κάθε εντολή. Το fmt κοιτάζει τη μορφή, το validate το κείμενο. Ορισμένα λάθη φαίνονται μόνο όταν το apply αγγίξει τον κόσμο.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform validate` ελέγχει σύνταξη, αναφορές, ονόματα και τύπους arguments. Χρειάζεται `init`, γιατί παίρνει τα schemas από τους providers. Δεν διαβάζει state, δεν χρειάζεται credentials και δεν καλεί κανένα API, άρα δεν πιάνει λάθη δικαιωμάτων ή αντικείμενα που λείπουν. Το `plan` και το `apply` κάνουν τον ίδιο έλεγχο αυτόματα πριν από οτιδήποτε άλλο. Σε CI χωρίς credentials το init γίνεται με `terraform init -backend=false`, που κατεβάζει τους providers χωρίς να συνδεθεί στο backend του κεφαλαίου 19.

ΚΡΑΤΑ ΑΥΤΟ

Πριν από κάθε commit τρέχεις `terraform fmt`, μετά `terraform validate`, μετά `terraform plan`. Το πρώτο τακτοποιεί, το δεύτερο πιάνει τα λάθη του κειμένου και το τρίτο δείχνει τι θα γίνει στον κόσμο.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch02`. Όπου χρειάζεται provider, γράψε το `versions.tf` του κεφαλαίου και τρέξε `terraform init` πριν από τις εντολές της εκφώνησης.

ΑΣΚΗΣΗ 2.1 Τακτοποίηση με fmt

Γράψε το παρακάτω `main.tf` όπως είναι. Τρέξε `terraform fmt -check`, μετά `terraform fmt` και δείξε το αποτέλεσμα με `cat main.tf`.

main.tf

```
resource "local_file" "menu"{
filename = "menu.txt"
      content="σαλάτα\nψάρι\n"   # σημερινό
  file_permission="0640"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
resource "local_file" "menu" {
  filename        = "menu.txt"
  content         = "σαλάτα\nψάρι\n" # σημερινό
  file_permission = "0640"
}
```

ΑΣΚΗΣΗ 2.2 Heredoc χωρίς περιττή εσοχή

Γράψε ένα `local_file` με όνομα `haiku`, που γράφει στο `haiku.txt` τις τρεις γραμμές του αναμενόμενου αποτελέσματος. Χρησιμοποίησε heredoc, με τις γραμμές στον κώδικα στοιχισμένες σε τέσσερα κενά. Στο αρχείο η δεύτερη γραμμή έχει δύο κενά εσοχή και οι άλλες καμία. Κάνε apply και δείξε το αρχείο με `cat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Φθινόπωρο.
  Ένα φύλλο πέφτει
στο τερματικό.
```

ΑΣΚΗΣΗ 2.3 Κείμενο με ειδικούς χαρακτήρες

Γράψε ένα `local_file` με όνομα `template`, που φτιάχνει το `template.txt` με ακριβώς τις δύο γραμμές του αναμενόμενου αποτελέσματος. Χρησιμοποίησε string σε εισαγωγικά, όχι heredoc. Κάνε apply και δείξε το αρχείο με `cat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Γεια σου, ${name}!
Είπε "ναι" στο C:\tf
```

ΑΣΚΗΣΗ 2.4 Το ίδιο όνομα σε δύο αρχεία

Γράψε το `versions.tf` του κεφαλαίου και ένα `a.tf` με ένα `local_file` με όνομα `note`, που φτιάχνει το `a.txt`. Τρέξε `terraform init`. Μετά πρόσθεσε ένα `b.tf` με ένα `local_file`, επίσης με όνομα `note`, που φτιάχνει το `b.txt`. Τρέξε `terraform validate`. Ποια αρχεία αναφέρει το μήνυμα;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Duplicate resource "local_file" configuration
╷
│   on b.tf line 1:
╷
│ A local_file resource named "note" was already declared at a.tf:1,1-29.
╵
```

ΑΣΚΗΣΗ 2.5 Διόρθωση σε γύρους

Γράψε το `versions.tf` του κεφαλαίου και το παρακάτω `main.tf`. Τρέξε `terraform init` και `terraform validate`. Διόρθωσε τα λάθη και ξανατρέξε validate μέχρι να δεις `Success!`. Το `toc` πρέπει να αναφέρει το `filename` του `chapter`. Για δικαιώματα φακέλου βάλε `0755`.

main.tf

```
resource "local_file" "chapter" {
  filname = "chapter.txt"
  content = "Κεφάλαιο 2\n"
}

resource "local_file" "toc" {
  filename             = "toc.txt"
  content              = "Περιεχόμενα του ${local_file.chapters.filename}\n"
  directory_permission = "rwxr-xr-x"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Success! The configuration is valid.
```

ΑΣΚΗΣΗ 2.6 Το validate λέει ναι, το apply όχι

Γράψε το `versions.tf` του κεφαλαίου και ένα `local_file` με όνομα `copy`, που φτιάχνει το `copy.txt`. Αντί για `content` δώσε `source = "original.txt"`, που λέει στον provider να αντιγράψει υπάρχον αρχείο. Το `original.txt` δεν υπάρχει. Τρέξε `terraform validate` και μετά `terraform apply` με απάντηση `yes`. Ποια εντολή εντοπίζει το πρόβλημα;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Success! The configuration is valid.
╷
│ +Original Error: open original.txt: no such file or directory
╵
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 2.1

Μια ομάδα θέλει το CI pipeline να αποτυγχάνει όταν κάποιο αρχείο `.tf` του repository, σε οποιονδήποτε φάκελο, δεν έχει την κανονική μορφή. Τα αρχεία δεν πρέπει να αλλάζουν. Ποια εντολή ταιριάζει;

a. `terraform fmt`
b. `terraform fmt -check -recursive`
c. `terraform validate`
d. `terraform fmt -diff`

ΕΡΩΤΗΣΗ 2.2

Ποιο από τα παρακάτω προβλήματα εντοπίζει το `terraform validate`;

a. Ο χρήστης δεν έχει δικαίωμα να δημιουργήσει το resource στον πάροχο.
b. Ένα argument μέσα σε resource έχει ορθογραφικό λάθος στο όνομά του.
c. Το όνομα που ζητήθηκε για ένα αντικείμενο το έχει ήδη πάρει άλλος.
d. Κάποιος άλλαξε με το χέρι ένα resource μετά το τελευταίο apply.

ΕΡΩΤΗΣΗ 2.3

Ποιες δύο προτάσεις για τα αρχεία μιας configuration είναι σωστές; Επίλεξε δύο.

a. Το Terraform διαβάζει όλα τα αρχεία `.tf` του φακέλου εργασίας ως μία configuration.
b. Το block `terraform` πρέπει να βρίσκεται σε αρχείο με όνομα `main.tf`.
c. Τα αρχεία `.tf` των υποφακέλων διαβάζονται αυτόματα.
d. Η σειρά των blocks και των αρχείων δεν επηρεάζει το αποτέλεσμα.
e. Ένα resource μπορεί να αναφέρεται μόνο σε resources του ίδιου αρχείου.

ΕΡΩΤΗΣΗ 2.4

Σωστό ή λάθος; Το `terraform validate` χρειάζεται credentials για τον πάροχο, για παράδειγμα κλειδιά του AWS, για να ελέγξει μια configuration.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 2.5

Τι κάνει το `terraform fmt` σε ένα αρχείο `.tf`;

a. Μετονομάζει τα resources σε snake_case.
b. Διορθώνει εσοχές και κενά και στοιχίζει τα `=` διαδοχικών arguments.
c. Διορθώνει ονόματα arguments με ορθογραφικά λάθη.
d. Ελέγχει ότι οι providers της configuration είναι εγκατεστημένοι.



ΚΕΦΑΛΑΙΟ 3

Το plan και οι αλλαγές

Το plan είναι το σημείο όπου πιάνεις ένα λάθος πριν γίνει ζημιά. Εδώ μαθαίνεις να διαβάζεις τα σύμβολά του και να ξεχωρίζεις την αλλαγή επί τόπου από την αντικατάσταση. Αποθηκεύεις ένα plan για να εφαρμοστεί ακριβώς αυτό και γνωρίζεις τις σημαίες του `plan` και του `apply` που ζητά η εξέταση.

Τι υπολογίζει το plan

Το plan συγκρίνει τη configuration με το state και με τα πραγματικά αντικείμενα. Το αποτέλεσμα είναι μια λίστα ενεργειών. Κάθε resource παίρνει το πολύ μία ενέργεια σε κάθε plan και κάθε ενέργεια έχει δικό της σύμβολο.

Σύμβολο	Ενέργεια	Πότε εμφανίζεται
`+`	create	Το resource υπάρχει στη configuration αλλά όχι στο state
`~`	update in place	Αλλάζει argument που ο provider αλλάζει στο υπάρχον αντικείμενο
`-`	destroy	Το resource υπάρχει στο state αλλά όχι πια στη configuration
`-/+`	replacement, πρώτα destroy και μετά create	Αλλάζει argument που απαιτεί νέο αντικείμενο
`+/-`	replacement, πρώτα create και μετά destroy	Όπως το `-/+`, με `create_before_destroy`

Το `+/-` εμφανίζεται μόνο με τη ρύθμιση `create_before_destroy`, που διδάσκεται στο κεφάλαιο 14. Τότε η επικεφαλίδα του plan γράφει `+/- create replacement and then destroy`. Εδώ αρκεί να το αναγνωρίζεις.

Η περίληψη `Plan: X to add, Y to change, Z to destroy.` μετρά ενέργειες. Ένα replacement σβήνει ένα αντικείμενο και φτιάχνει νέο. Μετρά λοιπόν μία φορά στο `to add` και μία στο `to destroy`.

Το resource terraform_data

Για να δεις update in place χρειάζεσαι resource που αλλάζει χωρίς να ξαναφτιαχτεί. Στον provider `local` κάθε αλλαγή ενός `local_file` σημαίνει νέο αρχείο, όπως είδες στο κεφάλαιο 1. Γι' αυτό χρησιμοποιούμε και το `terraform_data`.

Το `terraform_data` είναι ενσωματωμένο στο Terraform. Δεν κατεβαίνει από το Registry, οπότε δεν το δηλώνεις στο `required_providers`. Δεν φτιάχνει τίποτα έξω από το state. Απλώς κρατά τιμές, άρα μελετάς το plan χωρίς cloud. Μας ενδιαφέρουν δύο arguments.

- Το `input` αποθηκεύει μια τιμή, που μετά το apply εμφανίζεται και στο attribute `output`. Η αλλαγή του γίνεται επί τόπου.
- Το `triggers_replace` κρατά μια τιμή. Όταν αλλάξει, το resource αντικαθίσταται.

Σκέψου το σαν server. Το `input` είναι το όνομά του, που αλλάζει χωρίς νέα μηχανή. Το `triggers_replace` είναι το image του λειτουργικού, που θέλει νέα μηχανή. Το κεφάλαιο 6 εξηγεί το `terraform_data` αναλυτικά.

Το παράδειγμα του κεφαλαίου

TERMINAL

```
mkdir changes
cd changes
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "terraform_data" "server" {
  input            = "web-1"
  triggers_replace = "ubuntu-24.04"
}

resource "terraform_data" "cache" {
  input = "redis"
}

resource "local_file" "motd" {
  filename = "motd.txt"
  content  = "Καλώς ήρθες\n"
}
```

Το `terraform_data.server` είναι ο server μας, το `terraform_data.cache` ένας δεύτερος server για cache και το `local_file.motd` ένα αρχείο καλωσορίσματος.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- terraform.io/builtin/terraform is built in to Terraform
- Finding hashicorp/local versions matching "2.9.1"...
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
```

Η γραμμή `terraform.io/builtin/terraform is built in to Terraform` αφορά το `terraform_data`. Από το Registry κατέβηκε μόνο ο `local`.

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
  # local_file.motd will be created
…
  # terraform_data.cache will be created
…
  # terraform_data.server will be created
…
Plan: 3 to add, 0 to change, 0 to destroy.
…
  Enter a value: yes
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

Τρία create, η ερώτηση και το apply, όπως στο κεφάλαιο 1.

Update in place

Άλλαξε το όνομα του server από `web-1` σε `web-2`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

```
resource "terraform_data" "server" {
  input            = "web-2"
  triggers_replace = "ubuntu-24.04"
}

resource "terraform_data" "cache" {
  input = "redis"
}

resource "local_file" "motd" {
  filename = "motd.txt"
  content  = "Καλώς ήρθες\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
terraform_data.server: Refreshing state... [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
terraform_data.cache: Refreshing state... [id=66689c1d-b046-9e7b-9b36-323795b257ad]
local_file.motd: Refreshing state... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # terraform_data.server will be updated in-place
  ~ resource "terraform_data" "server" {
        id               = "6a5ed31f-1314-2c8a-7936-ebb4382f8ade"
      ~ input            = "web-1" -> "web-2"
      ~ output           = "web-1" -> (known after apply)
        # (1 unchanged attribute hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η επικεφαλίδα του plan αναφέρει μόνο το `~ update in-place`. Η γραμμή `# terraform_data.server will be updated in-place` λέει το ίδιο με λέξεις. Αυτή τη γραμμή πάνω από κάθε resource τη διαβάζεις πρώτη.

- Το `id` δεν έχει σύμβολο, γιατί δεν αλλάζει. Αυτή είναι η ουσία του update in place. Το αντικείμενο μένει το ίδιο.
- Το `input` γράφει `"web-1" -> "web-2"`, την τιμή του state αριστερά και τη νέα δεξιά.
- Το `output` θα αλλάξει, αλλά τη νέα τιμή θα την ορίσει το apply. Γι' αυτό γράφει `(known after apply)`.
- Το `# (1 unchanged attribute hidden)` κρύβει όσα δεν αλλάζουν, εδώ το `triggers_replace`.

Η περίληψη λέει `0 to add, 1 to change, 0 to destroy`. Τα resources χωρίς ενέργεια δεν εμφανίζονται καθόλου.



Replacement και forces replacement

Μην κάνεις ακόμη apply. Δώσε στο `motd.txt` δικαιώματα `0640` αντί για την προεπιλογή `0777` και σβήσε το block του cache server.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "terraform_data" "server" {
  input            = "web-2"
  triggers_replace = "ubuntu-24.04"
}

resource "local_file" "motd" {
  filename        = "motd.txt"
  content         = "Καλώς ήρθες\n"
  file_permission = "0640"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
terraform_data.cache: Refreshing state... [id=66689c1d-b046-9e7b-9b36-323795b257ad]
terraform_data.server: Refreshing state... [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
local_file.motd: Refreshing state... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place
```

```
  - destroy
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.motd must be replaced
-/+ resource "local_file" "motd" {
      …
      ~ file_permission      = "0777" -> "0640" # forces replacement
      ~ id                   = "82f813fe730a76d6254813b101d1cb3f5b5c9102" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

  # terraform_data.cache will be destroyed
  # (because terraform_data.cache is not in configuration)
  - resource "terraform_data" "cache" {
      - id     = "66689c1d-b046-9e7b-9b36-323795b257ad" -> null
      - input  = "redis" -> null
      - output = "redis" -> null
    }

  # terraform_data.server will be updated in-place
  ~ resource "terraform_data" "server" {
        id               = "6a5ed31f-1314-2c8a-7936-ebb4382f8ade"
      ~ input            = "web-1" -> "web-2"
      ~ output           = "web-1" -> (known after apply)
        # (1 unchanged attribute hidden)
    }

Plan: 1 to add, 1 to change, 2 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η επικεφαλίδα του plan έχει τώρα τρία σύμβολα, ένα για κάθε είδος ενέργειας.

Το `local_file.motd` έχει `-/+` και `must be replaced`. Το σχόλιο `# forces replacement` δίπλα στο `file_permission` δείχνει ποιο argument προκαλεί την αντικατάσταση. Ο provider `local` ορίζει ότι νέα δικαιώματα θέλουν νέο αρχείο, οπότε το παλιό θα σβηστεί και θα γραφτεί καινούργιο.

Το `terraform_data.cache` έχει `-`. Η γραμμή `(because terraform_data.cache is not in configuration)` εξηγεί ότι το block έφυγε από τη configuration. Κάθε attribute γράφει `-> null`, γιατί μετά το apply δεν θα υπάρχει.

Η περίληψη `1 to add, 1 to change, 2 to destroy` μετρά τέσσερις ενέργειες σε τρία resources. Το replacement του `motd` μετρά στο `to add` και στο `to destroy`. Το destroy του `cache` προσθέτει ένα ακόμη `to destroy`.

> **ΠΑΓΙΔΑ**
> Σε πραγματικό server το `-/+` σημαίνει ότι η μηχανή σβήνεται. Ο δίσκος της χάνεται, η IP αλλάζει και η υπηρεσία σταματά μέχρι να σηκωθεί η νέα. Όταν περιμένεις `~` και βλέπεις `-/+`, σταμάτα και βρες το `# forces replacement`.

Τι σημαίνει το (known after apply)

Το `(known after apply)` σημαίνει ότι η τιμή δεν υπολογίζεται στο plan. Θα την ορίσει ο provider όταν εκτελέσει την ενέργεια. Στο plan παραπάνω το έχει το `output` του server, που θα πάρει τιμή στο apply. Το έχει και το `id` του `motd`, που ανήκει σε αντικείμενο που δεν υπάρχει ακόμη. Η ένδειξη δεν σημαίνει ότι η τιμή θα αλλάξει οπωσδήποτε. Σημαίνει ότι το Terraform δεν την ξέρει από πριν.

Το apply χωρίς αποθηκευμένο plan

Κάθε plan μέχρι εδώ τελείωνε με τη σημείωση `You didn't use the -out option to save this plan`. Το plan δεν κρατήθηκε. Το σκέτο `terraform apply` υπολογίζει νέο plan από την αρχή, το δείχνει και ρωτά. Αν στο μεταξύ κάποιος έκανε apply, το νέο plan διαφέρει από αυτό που διάβασες. Γι' αυτό υπάρχει η ερώτηση, που δέχεται μόνο `yes`.

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 1 to change, 2 to destroy.
…
  Enter a value: no
…
Apply cancelled.
```

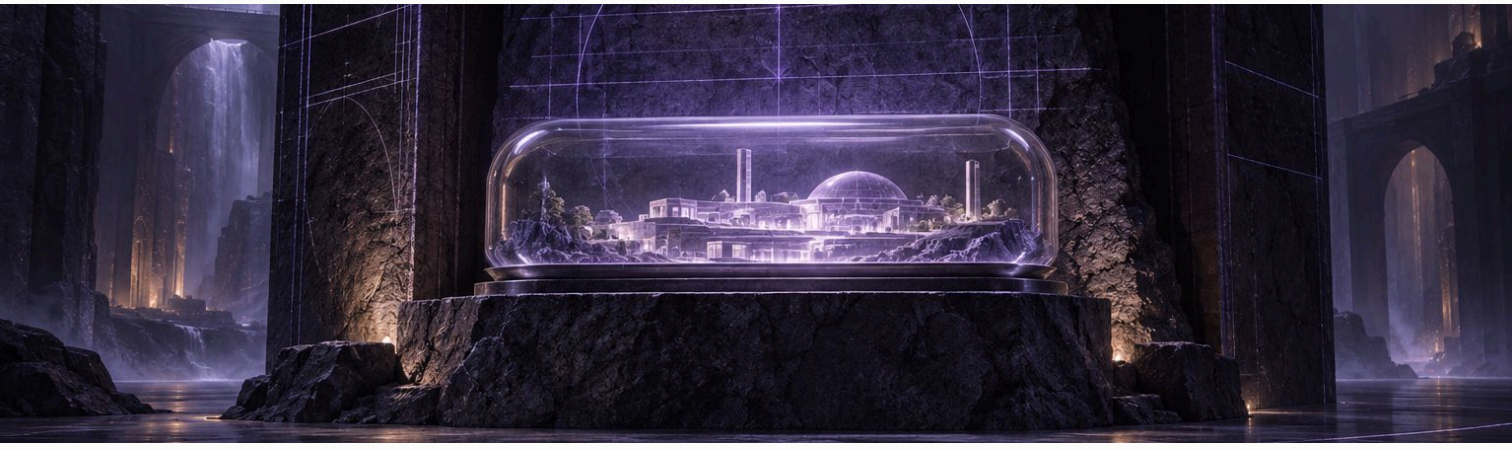
Το Terraform ακύρωσε χωρίς να αλλάξει τίποτα, όπως θα έκανε και με `Yes` ή σκέτο Enter. Το exit code δεν είναι μηδέν, οπότε και ένα script καταλαβαίνει ότι δεν έγινε apply.

Αποθηκευμένο plan

Η σημαία `-out` γράφει το plan σε αρχείο με όνομα της επιλογής σου. Η κατάληξη `.tfplan` είναι απλώς σύμβαση.

TERMINAL

```
terraform plan -out=changes.tfplan
```



ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 1 to change, 2 to destroy.

──────────────────────────────────────────────────────────────────────────────

Saved the plan to: changes.tfplan

To perform exactly these actions, run the following command to apply:
    terraform apply "changes.tfplan"
```

Το plan είναι ίδιο με πριν. Άλλαξε μόνο η σημείωση, που λέει πού αποθηκεύτηκε και πώς εφαρμόζεται. Το αρχείο το διαβάζεις με `terraform show`.

TERMINAL

```
terraform show changes.tfplan
```

ΕΞΟΔΟΣ

```
…
  # local_file.motd must be replaced
…
  # terraform_data.cache will be destroyed
  # (because terraform_data.cache is not in configuration)
…
  # terraform_data.server will be updated in-place
…
Plan: 1 to add, 1 to change, 2 to destroy.
```

Με αρχείο plan το `terraform show` δείχνει το plan όπως το `terraform plan`. Με `-json` το δίνει σε JSON για αυτόματους ελέγχους. Χωρίς όρισμα δείχνει το state, όπως θα δεις στο κεφάλαιο 4. Τώρα εφάρμοσε το αρχείο.

TERMINAL

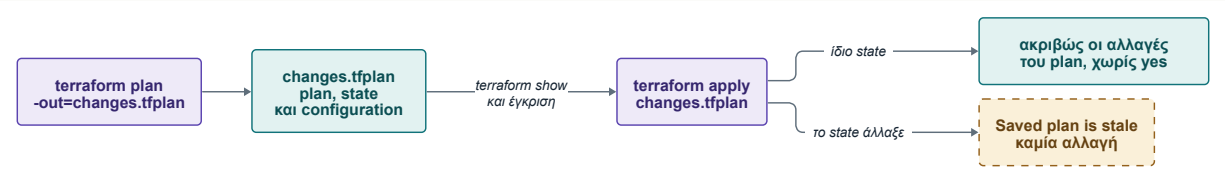
```
terraform apply changes.tfplan
```

ΕΞΟΔΟΣ

```
terraform_data.cache: Destroying... [id=66689c1d-b046-9e7b-9b36-323795b257ad]
terraform_data.cache: Destruction complete after 0s
terraform_data.server: Modifying... [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
terraform_data.server: Modifications complete after 0s [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
local_file.motd: Destroying... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]
local_file.motd: Destruction complete after 0s
local_file.motd: Creating...
local_file.motd: Creation complete after 0s [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Apply complete! Resources: 1 added, 1 changed, 2 destroyed.
```

Δεν υπάρχει ερώτηση. Το αρχείο θεωρείται εγκεκριμένο, αφού το διάβασες. Το Terraform δεν υπολογίζει νέο plan. Εκτελεί ακριβώς τις ενέργειες του αρχείου. Εδώ φαίνεται και το `-/+` στην πράξη. Το `local_file.motd` πρώτα σβήνεται (`Destroying`) και μετά ξαναφτιάχνεται (`Creating`). Το νέο `id` βγήκε ίδιο, γιατί είναι checksum του περιεχομένου, που δεν άλλαξε.

Το αρχείο κρατά και αντίγραφο της configuration. Αν αλλάξεις το `main.tf` μετά το `plan -out`, το apply του αρχείου αγνοεί την αλλαγή. Θα τη δεις στο επόμενο plan.



Σχήμα 3.1 · Το αποθηκευμένο plan. Το αρχείο κρατά το plan μαζί με το state και τη configuration από τις οποίες προέκυψε. Το apply του αρχείου δεν ρωτά και εφαρμόζεται μόνο αν το state δεν άλλαξε στο μεταξύ.

Τι περιέχει το αρχείο

TERMINAL

```
unzip -Z1 changes.tfplan
```

ΕΞΟΔΟΣ

```
tfplan
tfstate
tfstate-prev
tfconfig/m-/main.tf
tfconfig/modules.json
.terraform.lock.hcl
```

Το plan είναι δυαδικό αρχείο, ένα zip με εσωτερική δομή που μπορεί να αλλάξει από έκδοση σε έκδοση. Δεν το ανοίγεις με editor. Η λίστα όμως δείχνει ότι μέσα υπάρχουν το state (`tfstate`) και η configuration (`tfconfig`). Κωδικοί και κλειδιά που διαχειρίζεται η configuration βρίσκονται εκεί χωρίς κρυπτογράφηση, όπως στο state. Το `terraform show` κρύβει τις τιμές που δηλώνονται sensitive (κεφάλαιο 24), το αρχείο όμως τις έχει.

> **ΠΑΓΙΔΑ**
> Ένα plan αρχείο δεν μπαίνει σε Git. Το προστατεύεις όπως το state και το σβήνεις όταν τελειώσεις.

Stale plan

Το αρχείο ξέρει από ποιο state προέκυψε. Αν το state αλλάξει πριν από το apply, το plan δεν ισχύει πια. Δοκίμασε να εφαρμόσεις το ίδιο αρχείο δεύτερη φορά.

TERMINAL

```
terraform apply changes.tfplan
```

ΕΞΟΔΟΣ

```
Error: Saved plan is stale

The given plan file can no longer be applied because the state was changed
by another operation after the plan was created.
```

Το πρώτο apply άλλαξε το state, οπότε το αρχείο είναι πλέον stale και το Terraform αρνείται να το εφαρμόσει. Το ίδιο συμβαίνει όταν δύο άνθρωποι φτιάχνουν plans από το ίδιο state και ο ένας κάνει apply πρώτος. Η λύση είναι ένα νέο `terraform plan -out`.

ΚΡΑΤΑ ΑΥΤΟ

Σκέτο `terraform apply` σημαίνει νέο plan και ερώτηση. Το `terraform apply` με αρχείο σημαίνει ακριβώς το plan που αποθηκεύτηκε, χωρίς ερώτηση. Αυτό δουλεύει μόνο αν το state δεν άλλαξε από τη στιγμή του plan.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά τι κάνει το `terraform apply` με αρχείο plan. Δεν ζητά επιβεβαίωση, άρα δεν χρειάζεται `-auto-approve`. Ένα τέτοιο αρχείο το διαβάζεις με `terraform show`, γιατί είναι δυαδικό.

Χωρίς ερώτηση με -auto-approve

Η σημαία `-auto-approve` παραλείπει την ερώτηση του `apply` και του `destroy`. Δοκίμασέ τη αλλάζοντας το image του server.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "terraform_data" "server" {
  input            = "web-2"
  triggers_replace = "ubuntu-26.04"
}

resource "local_file" "motd" {
  filename        = "motd.txt"
  content         = "Καλώς ήρθες\n"
  file_permission = "0640"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
terraform_data.server: Refreshing state... [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
local_file.motd: Refreshing state... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # terraform_data.server must be replaced
```

```
-/+ resource "terraform_data" "server" {
      ~ id               = "6a5ed31f-1314-2c8a-7936-ebb4382f8ade" -> (known after apply)
      ~ output           = "web-2" -> (known after apply)
      ~ triggers_replace = "ubuntu-24.04" -> "ubuntu-26.04"
        # (1 unchanged attribute hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.
terraform_data.server: Destroying... [id=6a5ed31f-1314-2c8a-7936-ebb4382f8ade]
terraform_data.server: Destruction complete after 0s
terraform_data.server: Creating...
terraform_data.server: Creation complete after 0s [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Ο server έχει `-/+` και `must be replaced`, γιατί άλλαξε το `triggers_replace`. Το `# forces replacement` όμως λείπει. Το `triggers_replace` δέχεται τιμή οποιουδήποτε τύπου, όπως θα δεις στο κεφάλαιο 6. Σε τέτοια attributes το Terraform δεν τυπώνει το σχόλιο, ακόμη κι όταν προκαλούν την αντικατάσταση. Την αντικατάσταση λοιπόν τη διαβάζεις από το `-/+` και από τη γραμμή πάνω από το resource. Το `# forces replacement` σε βοηθά να βρεις την αιτία, αλλά δεν εμφανίζεται πάντα. Το `id` στη γραμμή `Creation complete` διαφέρει από αυτό στη γραμμή `Destroying`, γιατί ήρθε νέο αντικείμενο.

Το `-auto-approve` έχει θέση σε δύο περιπτώσεις.

- Σε περιβάλλοντα δοκιμών που φτιάχνεις και σβήνεις συχνά, όπως τα παραδείγματα του βιβλίου. Από εδώ και πέρα το χρησιμοποιούμε όπου η ερώτηση δεν προσφέρει τίποτα.
- Σε pipelines όπου κάποιος ενέκρινε το plan σε προηγούμενο βήμα. Εκεί όμως καλύτερο είναι το `plan -out` με apply του αρχείου, που εκτελεί ακριβώς ό,τι εγκρίθηκε.

> **ΠΑΓΙΔΑ**
> Μη βάζεις `-auto-approve` σε κοινή ή παραγωγική υποδομή για να γλιτώσεις ένα `yes`. Η ερώτηση είναι η τελευταία ευκαιρία να προσέξεις ένα `-/+`.

Αντικατάσταση με -replace

Μερικές φορές ένα αντικείμενο χαλάει ενώ η configuration δεν άλλαξε. Το Terraform δεν βλέπει διαφορά, οπότε δεν προτείνει τίποτα. Η σημαία `-replace` ζητά αντικατάσταση ενός resource με βάση τη διεύθυνσή του, δηλαδή τύπο και όνομα.

TERMINAL

```
terraform apply -replace=local_file.motd -auto-approve
```

ΕΞΟΔΟΣ

```
terraform_data.server: Refreshing state... [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]
local_file.motd: Refreshing state... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.motd will be replaced, as requested
-/+ resource "local_file" "motd" {
```

```
      …
      ~ id                  = "82f813fe730a76d6254813b101d1cb3f5b5c9102" -> (known after apply)
        # (4 unchanged attributes hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.
local_file.motd: Destroying... [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]
local_file.motd: Destruction complete after 0s
local_file.motd: Creating...
local_file.motd: Creation complete after 0s [id=82f813fe730a76d6254813b101d1cb3f5b5c9102]

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Η γραμμή πάνω από το resource γράφει `will be replaced, as requested`. Η αντικατάσταση έγινε επειδή τη ζήτησες και όχι λόγω κάποιου argument, γι' αυτό δεν υπάρχει `# forces replacement`. Το `-replace` δουλεύει και στο `terraform plan`, για να δεις πρώτα τι θα γίνει.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Παλαιότερα το ίδιο γινόταν με το `terraform taint`, που σημαδεύει ένα resource στο state για αντικατάσταση στο επόμενο apply. Υπάρχει ακόμη, αλλά προτείνεται το `-replace`, που δείχνει την αντικατάσταση στο plan πριν αλλάξει οτιδήποτε. Το κεφάλαιο 14 επανέρχεται στο θέμα.

Μερικό apply με -target

Η σημαία `-target` περιορίζει το plan σε ένα resource και σε όσα αυτό χρειάζεται. Τα υπόλοιπα αγνοούνται, ακόμη κι αν έχουν αλλαγές. Άλλαξε το όνομα του server σε `web-3` και το κείμενο του `motd`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "terraform_data" "server" {
  input            = "web-3"
  triggers_replace = "ubuntu-26.04"
}

resource "local_file" "motd" {
  filename        = "motd.txt"
  content         = "Καλώς ήρθες ξανά\n"
  file_permission = "0640"
}
```

TERMINAL

```
terraform apply -target=terraform_data.server -auto-approve
```

ΕΞΟΔΟΣ

```
terraform_data.server: Refreshing state... [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # terraform_data.server will be updated in-place
  ~ resource "terraform_data" "server" {
        id               = "b28c0429-239c-7c8f-a2d3-d6514ea4b709"
      ~ input            = "web-2" -> "web-3"
      ~ output           = "web-2" -> (known after apply)
        # (1 unchanged attribute hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.
terraform_data.server: Modifying... [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]
terraform_data.server: Modifications complete after 0s [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]
╷
│ Warning: Resource targeting is in effect
│
│ You are creating a plan with the -target option, which means that the
│ result of this plan may not represent all of the changes requested by the
│ current configuration.
│
│ The -target option is not for routine use, and is provided only for
│ exceptional situations such as recovering from errors or mistakes, or when
│ Terraform specifically suggests to use it as part of an error message.
╵

╷
│ Warning: Applied changes may be incomplete
│
│ The plan was created with the -target option in effect, so some changes
│ requested in the configuration may have been ignored and the output values
│ may not be fully updated. Run the following command to verify that no other
│ changes are pending:
│     terraform plan
│
│ Note that the -target option is not suitable for routine use, and is
│ provided only for exceptional situations such as recovering from errors or
│ mistakes, or when Terraform specifically suggests to use it as part of an
│ error message.
╵

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Το plan περιέχει μόνο τον server και η αλλαγή στο `motd` αγνοήθηκε. Η προειδοποίηση `Resource targeting is in effect` λέει ότι το plan μπορεί να μη δείχνει όλες τις αλλαγές. Η `Applied changes may be incomplete` σε συμβουλεύει να τρέξεις `terraform plan` για να δεις τι έμεινε.

Και οι δύο λένε ότι το `-target` δεν είναι για καθημερινή χρήση. Είναι για εξαιρέσεις, όπως η διόρθωση μετά από ένα λάθος ή όταν το ίδιο το Terraform το προτείνει σε μήνυμα σφάλματος. Η configuration περιγράφει όλη την υποδομή. Αν το `-target` γίνει συνήθεια, μένουν αλλαγές σε εκκρεμότητα και κανείς δεν ξέρει ποιες.

Το plan μέσα σε scripts

-no-color

Το Terraform χρωματίζει την έξοδο με κωδικούς ANSI. Στο terminal γίνονται χρώματα. Σε αρχείο ή σε pipe μένουν μέσα στο κείμενο.

TERMINAL

```
terraform plan > plan.txt
grep '^Plan' plan.txt
grep 'Plan' plan.txt | cat -v
```

ΕΞΟΔΟΣ

```
^[[1mPlan:^[[0m ^[[0m1 to add, 0 to change, 1 to destroy.
```

Το πρώτο `grep` δεν βρήκε τίποτα. Το `cat -v` δείχνει γιατί, κάνοντας ορατούς τους αόρατους χαρακτήρες. Η γραμμή δεν ξεκινά με `Plan` αλλά με `^[[1m`, τον κωδικό για έντονα γράμματα.

TERMINAL

```
terraform plan -no-color > plan.txt
grep '^Plan' plan.txt
```

ΕΞΟΔΟΣ

```
Plan: 1 to add, 0 to change, 1 to destroy.
```

Βάλε `-no-color` όταν η έξοδος καταλήγει σε αρχείο, σε log ενός pipeline ή σε άλλη εντολή. Αλλιώς τα logs γεμίζουν σύμβολα και τα φίλτρα αποτυγχάνουν σιωπηλά.

-detailed-exitcode

Ένα script κοιτάζει το exit code. Χωρίς σημαίες το `terraform plan` επιστρέφει `0` όταν πετυχαίνει, με αλλαγές ή χωρίς. Με `-detailed-exitcode` οι κωδικοί γίνονται τρεις.

- `0` σημαίνει επιτυχία χωρίς αλλαγές.
- `1` σημαίνει σφάλμα.
- `2` σημαίνει επιτυχία με αλλαγές που περιμένουν apply.

Η αλλαγή στο `motd` εκκρεμεί ακόμη από το `-target`.

TERMINAL

```
terraform plan -detailed-exitcode > /dev/null
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
exit code 2
```

Κάνε apply και ρώτα ξανά.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
terraform plan -detailed-exitcode > /dev/null
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
exit code 0
```

Για το `1` αρκεί ένα σφάλμα, για παράδειγμα μια διεύθυνση χωρίς τύπο resource.

TERMINAL

```
terraform plan -detailed-exitcode -target=motd
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid target "motd"
│
│ Resource specification must include a resource type and name.
╵

For more help on using this command, run:
  terraform plan -help
exit code 1
```

Έτσι ένα pipeline ελέγχει τακτικά αν η υποδομή ταιριάζει με τον κώδικα και ειδοποιεί όταν βγει `2`. Η σημαία υπάρχει μόνο στο `terraform plan`.

Destroy ως plan και ως apply

Το `terraform destroy` είναι συντόμευση. Η ίδια λειτουργία υπάρχει ως σημαία `-destroy` στο `plan` και στο `apply`.

TERMINAL

```
terraform plan -destroy
```

ΕΞΟΔΟΣ

```
terraform_data.server: Refreshing state... [id=b28c0429-239c-7c8f-a2d3-d6514ea4b709]
local_file.motd: Refreshing state... [id=8ee9b2c3adfa174b721d5270f60c5436e504face]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # local_file.motd will be destroyed
  - resource "local_file" "motd" {
      - content              = <<-EOT
```

```
            Καλώς ήρθες ξανά
        EOT -> null
      …
      - directory_permission = "0777" -> null
      - file_permission      = "0640" -> null
      - filename             = "motd.txt" -> null
      - id                   = "8ee9b2c3adfa174b721d5270f60c5436e504face" -> null
    }

  # terraform_data.server will be destroyed
  - resource "terraform_data" "server" {
      - id               = "b28c0429-239c-7c8f-a2d3-d6514ea4b709" -> null
      - input            = "web-3" -> null
      - output           = "web-3" -> null
      - triggers_replace = "ubuntu-26.04" -> null
    }

Plan: 0 to add, 0 to change, 2 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `plan -destroy` δείχνει τι θα σβηστεί χωρίς να σβήσει τίποτα. Η γραμμή με το `because` λείπει, γιατί δεν άλλαξε η configuration. Ζήτησες εσύ να φύγουν όλα. Συνδυάζεται και με `-out`, όπως κάθε plan.

TERMINAL

```
terraform apply -destroy -auto-approve
```

ΕΞΟΔΟΣ

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

Το `terraform apply -destroy` κάνει ό,τι και το `terraform destroy`. Μόνο η τελευταία γραμμή λέει `Apply complete!` αντί για το `Destroy complete!` του κεφαλαίου 1.

Οι σημαίες σε μία σελίδα

Εντολή ή σημαία	Τι κάνει
`terraform plan -out=F`	Αποθηκεύει το plan στο αρχείο F
`terraform show F`	Δείχνει ένα αποθηκευμένο plan
`terraform apply F`	Εκτελεί το αποθηκευμένο plan χωρίς ερώτηση
`terraform apply -auto-approve`	Νέο plan και εκτέλεση χωρίς ερώτηση
`terraform plan -destroy`	Δείχνει τι θα σβήσει ένα destroy
`terraform apply -destroy`	Ίδιο με το `terraform destroy`
`-replace=ADDRESS`	Αντικαθιστά ένα resource χωρίς αλλαγή στον κώδικα

Εντολή ή σημαία	Τι κάνει
`-target=ADDRESS`	Περιορίζει το plan σε ένα resource, μόνο για εξαιρέσεις
`-detailed-exitcode`	Exit code 0 χωρίς αλλαγές, 1 σε σφάλμα, 2 με αλλαγές
`-no-color`	Έξοδος χωρίς κωδικούς χρώματος

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch03`, με το `main.tf` της εκφώνησης και `terraform init` πρώτα. Οι ασκήσεις χρησιμοποιούν μόνο `terraform_data`, οπότε δεν χρειάζεται `required_providers`.

ΑΣΚΗΣΗ 3.1 Ένα σύμβολο

Κάνε `terraform apply -auto-approve` με αυτό το `main.tf`.

main.tf

```
resource "terraform_data" "app" {
  input            = "v1"
  triggers_replace = "a"
}
```

Άλλαξε το `input` σε `"v2"`. Πριν τρέξεις `terraform plan`, γράψε ποιο σύμβολο και ποια περίληψη περιμένεις.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # terraform_data.app will be updated in-place
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 3.2 Δύο αλλαγές σε ένα resource

Κάνε apply με το ίδιο `main.tf` όπως στην προηγούμενη άσκηση.

main.tf

```
resource "terraform_data" "app" {
  input            = "v1"
  triggers_replace = "a"
}
```

Άλλαξε μαζί το `input` σε `"v2"` και το `triggers_replace` σε `"b"`. Θα δεις ένα `~` και ένα `-/+` ή κάτι άλλο;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # terraform_data.app must be replaced
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 3.3 Plan που αγνοεί τις νέες αλλαγές

Με αυτό το `main.tf` τρέξε `terraform plan -out=app.tfplan` χωρίς apply. Μετά άλλαξε το `input` σε `"v2"`, εφάρμοσε το `app.tfplan` και τρέξε `terraform plan`. Ποια τιμή κατέληξε στο state;

main.tf

```
resource "terraform_data" "app" {
  input = "v1"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      ~ input  = "v1" -> "v2"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 3.4 Κωδικοί εξόδου

Με αυτό το `main.tf` τρέξε `terraform plan -detailed-exitcode` πριν και μετά από ένα apply. Τύπωσε κάθε φορά το exit code με `echo`, για παράδειγμα `echo "πριν το apply $?"`.

main.tf

```
resource "terraform_data" "app" {
  input = "v1"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
πριν το apply 2
μετά το apply 0
```

ΑΣΚΗΣΗ 3.5 Η περίληψη του destroy σε αρχείο

Κάνε apply με αυτό το `main.tf`. Γράψε μετά σε αρχείο το plan που θα έσβηνε τα πάντα και κράτα μόνο την περίληψη με `grep '^Plan'`.

main.tf

```
resource "terraform_data" "a" {
  input = "a"
}

resource "terraform_data" "b" {
  input = "b"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Plan: 0 to add, 0 to change, 2 to destroy.
```

ΑΣΚΗΣΗ 3.6 Ξαναφτιάξε ένα resource

Κάνε apply με αυτό το `main.tf`. Αντικατάστησε μετά μόνο το `terraform_data.b`, χωρίς αλλαγή στον κώδικα και χωρίς ερώτηση.

main.tf

```
resource "terraform_data" "a" {
  input = "a"
}

resource "terraform_data" "b" {
  input = "b"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # terraform_data.b will be replaced, as requested
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 3.1

Στο plan βλέπεις τη γραμμή `-/+ resource "aws_instance" "web" {`. Τι θα κάνει το Terraform στο apply;

a. Θα ενημερώσει το υπάρχον instance χωρίς να το σβήσει.
b. Θα φτιάξει νέο instance και μετά θα σβήσει το παλιό.
c. Θα σβήσει το υπάρχον instance και μετά θα φτιάξει νέο.
d. Θα σβήσει το instance χωρίς να φτιάξει νέο.

ΕΡΩΤΗΣΗ 3.2

Ένα plan περιέχει ένα resource με `~`, ένα με `-/+` και ένα με `-`. Ποια είναι η περίληψη;

a. `Plan: 0 to add, 3 to change, 0 to destroy.`
b. `Plan: 1 to add, 1 to change, 2 to destroy.`
c. `Plan: 1 to add, 2 to change, 1 to destroy.`
d. `Plan: 0 to add, 1 to change, 1 to destroy.`

ΕΡΩΤΗΣΗ 3.3

Τρέχεις `terraform plan -out=tfplan` και μετά `terraform apply tfplan`. Τι συμβαίνει;

a. Εκτελούνται οι ενέργειες του αρχείου χωρίς ερώτηση.
b. Το Terraform φτιάχνει νέο plan από την τρέχουσα configuration και ζητά `yes`.
c. Το Terraform ζητά `yes`, εκτός αν προσθέσεις `-auto-approve`.
d. Το Terraform αποτυγχάνει, γιατί το αρχείο θέλει πρώτα `terraform show`.

ΕΡΩΤΗΣΗ 3.4

Ένα pipeline τρέχει `terraform plan -detailed-exitcode`. Τι σημαίνει exit code `2`;

a. Υπάρχει σφάλμα στη configuration.
b. Το plan πέτυχε και δεν υπάρχουν αλλαγές.
c. Το αποθηκευμένο plan είναι stale.
d. Το plan πέτυχε και υπάρχουν αλλαγές.

ΕΡΩΤΗΣΗ 3.5

Ποιες προτάσεις ισχύουν για το αρχείο που γράφει το `terraform plan -out`; Επίλεξε δύο.

a. Μπορεί να περιέχει ευαίσθητες τιμές χωρίς κρυπτογράφηση.
b. Είναι αρχείο κειμένου που μπορείς να διορθώσεις πριν από το apply.
c. Διαβάζεται με `terraform show`.
d. Εφαρμόζεται ακόμη κι αν το state άλλαξε μετά τη δημιουργία του.
e. Μπαίνει σε Git μαζί με τη configuration.

ΕΡΩΤΗΣΗ 3.6

Ένας server που διαχειρίζεται το Terraform έχει κολλήσει, ενώ η configuration δεν άλλαξε. Θέλεις να τον αντικαταστήσεις. Ποιος είναι ο προτεινόμενος τρόπος;

a. `terraform apply -target=aws_instance.web`
b. `terraform apply -replace=aws_instance.web`
c. Σβήνεις το resource από το `terraform.tfstate` και κάνεις apply.
d. `terraform destroy` και μετά `terraform apply`.



ΚΕΦΑΛΑΙΟ 4

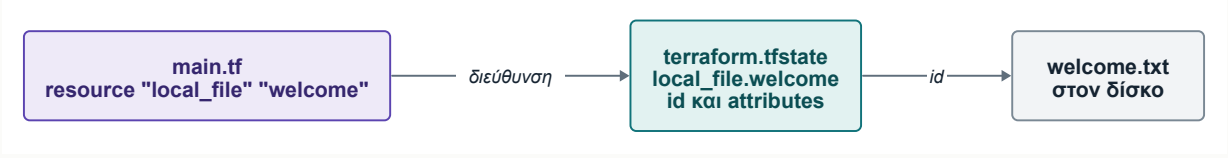
Το state

Μετά το πρώτο apply εμφανίστηκε δίπλα στη configuration το αρχείο `terraform.tfstate`. Σε αυτό το κεφάλαιο το ανοίγεις, καταλαβαίνεις γιατί το Terraform δεν δουλεύει χωρίς αυτό και βλέπεις πώς το ελέγχει σε κάθε plan. Μαθαίνεις επίσης πού αποθηκεύεται, τι μυστικά κρατά και γιατί δεν το αγγίζεις ποτέ με editor.

Γιατί χρειάζεται το state

Η configuration λέει τι θέλεις. Ο πραγματικός κόσμος έχει ό,τι υπάρχει. Το **state** είναι η μνήμη του Terraform που τα συνδέει. Για κάθε resource γράφει ποιο πραγματικό αντικείμενο του αντιστοιχεί και ποια attributes είχε την τελευταία φορά που το είδε. Γιατί όμως το Terraform δεν κοιτάζει απλώς την υποδομή κάθε φορά; Η απάντηση έχει τέσσερα σκέλη.

Αντιστοίχιση. Ένα resource `aws_instance.web` γίνεται στο AWS ένας server με ID σαν `i-0abc12345`. Το AWS δεν ξέρει ότι αυτός ο server ανήκει στη δική σου configuration. Στον ίδιο λογαριασμό μπορεί να υπάρχουν εκατό servers από άλλους ανθρώπους ή άλλες configurations. Τα ονόματα και τα tags δεν είναι ούτε υποχρεωτικά ούτε μοναδικά. Το state κρατά τη σύνδεση ένα προς ένα. Η διεύθυνση του resource δείχνει σε ένα συγκεκριμένο ID και σε κανένα άλλο.



Σχήμα 4.1 · Το state συνδέει τη διεύθυνση ενός resource της configuration με το id ενός πραγματικού αντικειμένου.

Metadata. Αν σβήσεις ένα block από τη configuration, το Terraform πρέπει να καταστρέψει το αντικείμενο που δεν αναφέρεται πια πουθενά. Μόνο το state θυμάται ότι υπάρχει, ποιος provider το διαχειρίζεται και από ποια resources εξαρτάται, ώστε να το σβήσει με τη σωστή σειρά. Τις εξαρτήσεις (dependencies) τις γνωρίζεις στο κεφάλαιο 6.

Απόδοση. Το state κρατά αντίγραφο όλων των attributes. Σε μια υποδομή με χιλιάδες resources το να ρωτάς το API για όλα σε κάθε plan κοστίζει λεπτά. Τότε μπορείς να ζητήσεις plan μόνο από όσα θυμάται το state, όπως δείχνει το κεφάλαιο 21. Κάποιες τιμές μάλιστα ζουν μόνο στο state. Ένα τυχαίο όνομα από τον provider `random` του κεφαλαίου 6 δεν υπάρχει σε κανένα cloud για να το ξαναδιαβάσεις.

Συνεργασία. Όταν δύο άνθρωποι δουλεύουν στην ίδια υποδομή, πρέπει να βλέπουν την ίδια μνήμη. Το state μπαίνει τότε σε κοινό χώρο, για παράδειγμα σε ένα S3 bucket. Εκεί κλειδώνεται όσο κάποιος κάνει αλλαγές. Αυτό είναι το θέμα του κεφαλαίου 19.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Η εξέταση ρωτά τον σκοπό του state. Η απάντηση που ψάχνει είναι ότι αντιστοιχίζει τα resources της configuration σε πραγματικά αντικείμενα και κρατά metadata όπως οι εξαρτήσεις. Ακολουθούν η απόδοση και η συνεργασία. Το state δεν είναι προαιρετικό. Το Terraform δεν δουλεύει χωρίς αυτό.

Ένα state για να το εξετάσεις

Φτιάξε έναν φάκελο για μια μικρή ομάδα με ένα αρχείο υποδοχής.

TERMINAL

```
mkdir team
cd team
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "welcome" {
  filename = "welcome.txt"
  content  = "Καλώς ήρθες στην ομάδα\n"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το `-auto-approve` του κεφαλαίου 3 παραλείπει την ερώτηση, γιατί το plan το ξέρεις ήδη. Τώρα άνοιξε το state. Τα checksums του περιεχομένου φαίνονται ως `…` για να χωρά η έξοδος.

TERMINAL

```
cat terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "version": 4,
  "terraform_version": "1.16.4",
  "serial": 1,
  "lineage": "36a89b07-5b86-52c4-1e35-49e4158292e3",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "local_file",
      "name": "welcome",
      "provider": "provider[\"registry.terraform.io/hashicorp/local\"]",
      "instances": [
        {
          "schema_version": 0,
          "attributes": {
            "content": "Καλώς ήρθες στην ομάδα\n",
            "content_base64": null,
            …
            "directory_permission": "0777",
            "file_permission": "0777",
            "filename": "welcome.txt",
            "id": "e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c",
            "sensitive_content": null,
            "source": null
          },
          "sensitive_attributes": [
            [
              {
                "type": "get_attr",
                "value": "sensitive_content"
              }
            ]
          ],
          "identity_schema_version": 0
        }
      ]
    }
  ],
  "check_results": null
}
```

Το αρχείο είναι JSON. Στο πρώτο επίπεδο βρίσκονται πεδία για όλο το state.

- `version` είναι η έκδοση της μορφής του αρχείου, όχι του Terraform. Η τρέχουσα μορφή είναι η 4.
- `terraform_version` είναι η έκδοση του Terraform που έγραψε το αρχείο τελευταία φορά.
- `serial` είναι αριθμός που μεγαλώνει όταν το Terraform γράφει αλλαγμένο state.

- `lineage` είναι μοναδικό αναγνωριστικό που πήρε το state όταν δημιουργήθηκε. Το δικό σου θα διαφέρει.
- `outputs` κρατά τις τιμές των outputs του κεφαλαίου 8. Εδώ είναι άδειο.
- `resources` είναι η λίστα με ό,τι διαχειρίζεται η configuration.
- `check_results` κρατά τα αποτελέσματα των ελέγχων του κεφαλαίου 13.

Κάθε στοιχείο του `resources` περιγράφει ένα block `resource`. Το `mode` είναι `managed`. Τα data sources του κεφαλαίου 12 γράφονται με `data`. Τα `type` και `name` είναι τα δύο labels του block και μαζί δίνουν τη διεύθυνση `local_file.welcome`. Το `provider` είναι η πλήρης διεύθυνση του provider στο Registry. Το `instances` είναι λίστα, γιατί με τα `count` και `for_each` του κεφαλαίου 11 ένα block φτιάχνει πολλά αντικείμενα.

Η καρδιά του state είναι το `attributes`. Περιέχει όσα έγραψες εσύ και όσα υπολόγισε ο provider. Όσα δεν ορίστηκαν, όπως το `sensitive_content`, είναι `null`. Το `id` είναι ο σύνδεσμος με τον πραγματικό κόσμο. Τα υπόλοιπα πεδία, όπως το `schema_version`, τα χρησιμοποιεί εσωτερικά το Terraform.

serial και lineage

Πρόσθεσε ένα δεύτερο αρχείο με τους κανόνες της ομάδας. Πριν από το apply σημείωσε τα δύο πεδία.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "welcome" {
  filename = "welcome.txt"
  content  = "Καλώς ήρθες στην ομάδα\n"
}

resource "local_file" "rules" {
  filename = "rules.txt"
  content  = "Πρώτα plan, μετά apply.\n"
}
```

TERMINAL

```
jq '{serial, lineage}' terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "serial": 1,
  "lineage": "36a89b07-5b86-52c4-1e35-49e4158292e3"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

TERMINAL

```
jq '{serial, lineage}' terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "serial": 3,
  "lineage": "36a89b07-5b86-52c4-1e35-49e4158292e3"
}
```

Το `serial` ανέβηκε από 1 σε 3 και το `lineage` έμεινε ίδιο.

Ένα apply δεν γράφει το state μόνο στο τέλος. Το γράφει και μετά από κάθε ενέργεια που ολοκληρώνει, ώστε αν διακοπεί στη μέση το αρχείο να λέει τι πρόλαβε να γίνει. Κάθε τέτοια εγγραφή μπορεί να ανεβάσει το `serial`. Εδώ το apply έγραψε μετά τη δημιουργία του `rules` και ξανά στο τέλος, οπότε το `serial` ανέβηκε κατά δύο για ένα μόνο resource. Πόσο ακριβώς ανεβαίνει σε κάθε apply είναι εσωτερική λεπτομέρεια του Terraform. Ένα apply χωρίς αλλαγές δεν το ανεβάζει καθόλου. Το `serial` λοιπόν δεν μετρά applies ούτε resources. Λέει ποιο από δύο αντίγραφα του ίδιου state είναι νεότερο.

Το `lineage` δίνεται τη στιγμή που δημιουργείται το state και δεν αλλάζει ποτέ. Δύο αρχεία με το ίδιο `lineage` είναι εκδοχές του ίδιου state. Δύο αρχεία με διαφορετικό `lineage` έχουν άλλη ιστορία, ακόμη κι αν περιγράφουν τα ίδια resources. Αν σβήσεις το state και ξανακάνεις apply, το νέο state παίρνει νέο `lineage`. Τα δύο πεδία εμποδίζουν να γραφτεί ένα παλιό ή ξένο state πάνω στο σωστό. Το `terraform state push` του κεφαλαίου 21 αρνείται να γράψει ένα state με άλλο `lineage` ή με μικρότερο `serial` από το τρέχον.

Το terraform.tfstate.backup

Μετά το δεύτερο apply εμφανίστηκε ένα ακόμη αρχείο.

TERMINAL

```
ls
jq '{serial, names: [.resources[].name]}' terraform.tfstate.backup
jq '{serial, names: [.resources[].name]}' terraform.tfstate
```

ΕΞΟΔΟΣ

```
main.tf
rules.txt
terraform.tfstate
terraform.tfstate.backup
welcome.txt
{
  "serial": 1,
  "names": [
    "welcome"
  ]
}
{
  "serial": 3,
  "names": [
    "rules",
    "welcome"
  ]
}
```

Το backup έχει `serial` 1 και μόνο το `welcome`. Είναι το state πριν από το τελευταίο apply. Την πρώτη φορά που μια εντολή γράφει το state, το Terraform αντιγράφει το παλιό αρχείο στο `terraform.tfstate.backup`. Οι επόμενες εγγραφές της ίδιας εντολής δεν το αγγίζουν. Μετά το πρώτο apply δεν υπήρχε backup, γιατί δεν υπήρχε παλιό αρχείο.

Το backup κρατά μόνο ένα βήμα πίσω και δεν είναι ιστορικό. Ένα plan ή ένα apply χωρίς αλλαγές δεν το αγγίζουν. Αν χαθεί το `terraform.tfstate`, το Terraform δεν διαβάζει μόνο του το backup. Το αρχείο αυτό το γράφει το local backend, που θα δεις παρακάτω.

Εντολές που διαβάζουν το state

Τα `cat` και `jq` δουλεύουν, αλλά το Terraform έχει δικές του εντολές, που δουλεύουν με κάθε backend. Η `terraform state list` τυπώνει τις διευθύνσεις όλων των resources του state.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
local_file.rules
local_file.welcome
```

Η `terraform state show` δείχνει ένα resource με όλα τα attributes του, σε μορφή που μοιάζει με HCL.

TERMINAL

```
terraform state show local_file.rules
```

ΕΞΟΔΟΣ

```
# local_file.rules:
resource "local_file" "rules" {
    content              = <<-EOT
        Πρώτα plan, μετά apply.
    EOT
    …
    directory_permission = "0777"
    file_permission      = "0777"
    filename             = "rules.txt"
    id                   = "5709f6ccf9c96c112ca7b3b252f703113f21fbe1"
}
```

Η πρώτη γραμμή είναι η διεύθυνση σε σχόλιο. Ακολουθούν τα attributes που είδες στο JSON. Το `terraform show` χωρίς όρισμα τυπώνει με τον ίδιο τρόπο όλα τα resources. Στο κεφάλαιο 3 το χρησιμοποίησες για ένα αποθηκευμένο plan. Χωρίς όρισμα δείχνει το τρέχον state.

TERMINAL

```
terraform show
```

ΕΞΟΔΟΣ

```
# local_file.rules:
…
# local_file.welcome:
…
```

Για scripts υπάρχει το `terraform show -json`. Δίνει το state σε JSON με τεκμηριωμένη μορφή, που προορίζεται για προγράμματα.

TERMINAL

```
terraform show -json | jq '.values.root_module.resources[] | {address, file: .values.filename}'
```

ΕΞΟΔΟΣ

```
{
  "address": "local_file.rules",
  "file": "rules.txt"
}
{
  "address": "local_file.welcome",
  "file": "welcome.txt"
}
```

Τα resources βρίσκονται κάτω από το `values.root_module`, γιατί ανήκουν στο root module. Τα child modules του κεφαλαίου 15 έχουν δικό τους κλαδί.

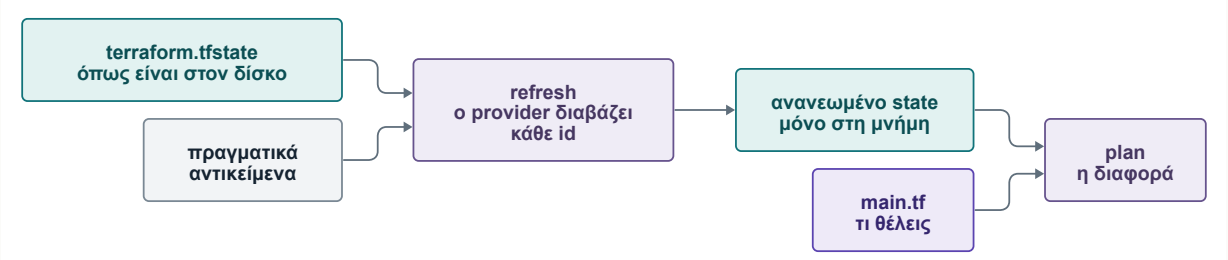
ΚΡΑΤΑ ΑΥΤΟ

Για να δεις τι θυμάται το Terraform, χρησιμοποίησε `terraform state list`, `terraform state show` και `terraform show`. Για scripts χρησιμοποίησε το `terraform show -json`. Η μορφή του `terraform.tfstate` είναι εσωτερική και μπορεί να αλλάξει.



Refresh μέσα στο plan

Κάθε plan ξεκινά με **refresh**. Για κάθε resource του state το Terraform ζητά από τον provider να διαβάσει το πραγματικό αντικείμενο με το `id` του. Από τις απαντήσεις φτιάχνει ένα ανανεωμένο αντίγραφο του state στη μνήμη και συγκρίνει τη configuration με αυτό. Οι γραμμές `Refreshing state...` είναι αυτό το βήμα. Το plan δεν γράφει το ανανεωμένο αντίγραφο στον δίσκο.



Σχήμα 4.2 · Στο refresh ο provider διαβάζει τα πραγματικά αντικείμενα. Το plan συγκρίνει τη configuration με το ανανεωμένο state, που μένει στη μνήμη.

Σβήσε το `welcome.txt` έξω από το Terraform, όπως θα το έσβηνε από λάθος ένας συνάδελφος. Μετά τρέξε plan.

TERMINAL

```
rm welcome.txt
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.welcome: Refreshing state... [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
local_file.rules: Refreshing state... [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
```

```
  # local_file.welcome will be created
  + resource "local_file" "welcome" {
      + content              = <<-EOT
            Καλώς ήρθες στην ομάδα
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "welcome.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Στο refresh ο provider έψαξε το `welcome.txt` και δεν το βρήκε. Το ανανεωμένο state δεν έχει πια το `local_file.welcome`, ενώ η configuration το ζητά, οπότε το plan προτείνει να δημιουργηθεί. Μια αλλαγή που έγινε έξω από το Terraform λέγεται **drift**. Το refresh είναι ο τρόπος που τη βρίσκει.

TERMINAL

```
jq .serial terraform.tfstate
terraform state list
```

ΕΞΟΔΟΣ

```
3
local_file.rules
local_file.welcome
```

Το αρχείο δεν άλλαξε. Το `serial` είναι ακόμη 3 και το state θυμάται και τα δύο resources. Το apply κάνει το ίδιο refresh, ξαναφτιάχνει το αρχείο και γράφει το νέο state.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
local_file.welcome: Refreshing state... [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
…
local_file.welcome: Creating...
local_file.welcome: Creation complete after 0s [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Δοκίμασε τώρα μια πιο ύπουλη αλλαγή. Γράψε άλλο κείμενο μέσα στο `rules.txt`.

TERMINAL

```
echo "Κανένας κανόνας" > rules.txt
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.welcome: Refreshing state... [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
local_file.rules: Refreshing state... [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.rules will be created
  + resource "local_file" "rules" {
      + content              = <<-EOT
            Πρώτα plan, μετά apply.
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "rules.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το αρχείο υπάρχει, αλλά το plan θέλει πάλι να το δημιουργήσει, με `+` και όχι με `~`. Στο refresh ο provider `local` υπολογίζει το checksum του αρχείου στον δίσκο και το συγκρίνει με το `id` του state. Το `id` ενός `local_file` είναι το checksum του περιεχομένου που έγραψε, όπως είδες στο κεφάλαιο 3. Όταν τα δύο διαφέρουν, ο provider θεωρεί ότι το αρχείο του χάθηκε και το βγάζει από το ανανεωμένο state. Το plan λοιπόν το βλέπει ως νέο και το ξαναγράφει.

Άλλοι providers διαβάζουν κάθε attribute χωριστά. Ο provider του AWS βρίσκει ποιο attribute ενός server άλλαξε και το plan δείχνει μόνο αυτή τη διαφορά. Κάθε provider αποφασίζει τι διαβάζει στο refresh. Ό,τι δεν διαβάζει, το Terraform δεν το βλέπει. Ο `local` για παράδειγμα δεν ελέγχει τα δικαιώματα. Ένα `chmod` στο `rules.txt` δεν θα εμφανιζόταν στο plan.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Τα `terraform plan` και `terraform apply` κάνουν refresh από προεπιλογή. Το plan δεν γράφει το αποτέλεσμα στο state. Η παλιά εντολή `terraform refresh` γράφει το ανανεωμένο state χωρίς να δείξει τι αλλάζει και είναι πλέον deprecated. Τη θέση της πήρε το `terraform apply -refresh-only`, που δείχνει πρώτα τις αλλαγές και ζητά έγκριση. Το κεφάλαιο 21 το εξηγεί μαζί με το `-refresh=false`.

Άφησε το `rules.txt` όπως είναι. Θα χρειαστεί αμέσως παρακάτω.

To local backend

Το **backend** αποφασίζει πού αποθηκεύεται το state και πώς κλειδώνεται. Όταν η configuration δεν ορίζει backend, το Terraform χρησιμοποιεί το **local backend**, που γράφει το `terraform.tfstate` στον φάκελο εργασίας. Αυτό έκανε σε όλα τα παραδείγματα ως τώρα. Η ενότητα `Initializing the backend` του `terraform init` αφορά αυτή την επιλογή.

To lock

Αν δύο εντολές γράψουν το ίδιο state ταυτόχρονα, η μία σβήνει τη δουλειά της άλλης. Γι' αυτό κάθε plan, apply και destroy κλειδώνει πρώτα το state. Το local backend βάζει lock στο αρχείο μέσω του λειτουργικού συστήματος και γράφει δίπλα το `.terraform.tfstate.lock.info`. Όταν η εντολή τελειώσει, σβήνει το αρχείο και ελευθερώνει το lock.

Ένα apply που περιμένει απάντηση κρατά το lock όσο περιμένει. Με δεύτερο terminal θα το άφηνες να περιμένει στο ένα και θα συνέχιζες στο άλλο. Εδώ το apply τρέχει στο παρασκήνιο, γιατί η γραμμή τελειώνει σε `&`. Το `(sleep 60; echo no)` του απαντά `no` μετά από ένα λεπτό και η έξοδός του πηγαίνει στο `apply.log`. Σε δικό σου terminal το shell τυπώνει επιπλέον μια γραμμή σαν `[1] 4321` με τον αριθμό της εργασίας. Δεν αφορά το Terraform.

TERMINAL

```
(sleep 60; echo no) | terraform apply > apply.log 2>&1 &
sleep 3
ls -a
```

ΕΞΟΔΟΣ

```
.
..
.terraform
.terraform.lock.hcl
.terraform.tfstate.lock.info
apply.log
main.tf
rules.txt
terraform.tfstate
terraform.tfstate.backup
welcome.txt
```

Το apply βρήκε την αλλαγή στο `rules.txt` και περιμένει απάντηση. Δίπλα στο state εμφανίστηκε το `.terraform.tfstate.lock.info`. Τρέξε τώρα ένα plan, πριν περάσει το λεπτό.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Error acquiring the state lock
│
│ Error message: resource temporarily unavailable
│ Lock Info:
│   ID:        fb0c7a57-4ce0-c064-b2da-6fc49587df33
│   Path:      terraform.tfstate
│   Operation: OperationTypeApply
│   Who:       you@laptop
│   Version:   1.16.4
│   Created:   2026-10-08 02:26:27.339744 +0000 UTC
│   Info:
│
│
│ Terraform acquires a state lock to protect the state from being written
│ by multiple users at the same time. Please resolve the issue above and try
│ again. For most commands, you can disable locking with the "-lock=false"
│ flag, but this is not recommended.
╵
```

Το plan σταμάτησε αμέσως. Το `Lock Info` λέει ποιο αρχείο είναι κλειδωμένο, από ποια εντολή, ποιος χρήστης σε ποιον υπολογιστή και από πότε. Το ID και η ώρα αλλάζουν σε κάθε εκτέλεση. Το μήνυμα αναφέρει το `-lock=false`, που παρακάμπτει το lock. Μην το χρησιμοποιείς. Υπάρχει για σπάνιες περιπτώσεις όπου ξέρεις ότι κανείς άλλος δεν γράφει. Το `wait` περιμένει να τελειώσει το apply του παρασκηνίου.

TERMINAL

```
wait
tail -1 apply.log
rm apply.log
ls -a
```

ΕΞΟΔΟΣ

```
Apply cancelled.
.
..
.terraform
.terraform.lock.hcl
main.tf
rules.txt
terraform.tfstate
terraform.tfstate.backup
welcome.txt
```

Μόλις το apply πήρε το `no`, ακυρώθηκε, έσβησε το lock info και ελευθέρωσε το lock. Το lock ενός τοπικού αρχείου προστατεύει μόνο εντολές στον ίδιο υπολογιστή. Για ομάδες χρειάζεται remote backend με κοινό lock. Αυτό, μαζί με το `terraform force-unlock` για locks που έμειναν ορφανά, είναι θέμα του κεφαλαίου 19. Τώρα κάνε το apply που ακυρώθηκε, για να διορθωθεί το `rules.txt`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
local_file.rules: Refreshing state... [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]
…
local_file.rules: Creating...
local_file.rules: Creation complete after 0s [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Άλλο path για το state

Με ένα block `backend "local"` μέσα στο block `terraform` ορίζεις άλλο path για το state. Φτιάξε δεύτερο φάκελο δίπλα στο `team`.

TERMINAL

```
cd ..
mkdir notes
cd notes
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }

  backend "local" {
    path = "state/notes.tfstate"
  }
}

resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Σημειώσεις της ομάδας\n"
}
```

Το label του block `backend` είναι ο τύπος του backend. Το `path` είναι σχετικό με τον φάκελο εργασίας.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Successfully configured the backend "local"! Terraform will automatically
use this backend unless the backend configuration changes.
…
```

Αυτή τη φορά το `init` λέει ότι ρύθμισε το backend `"local"`.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
find . -name "*.tfstate" | sort
```

ΕΞΟΔΟΣ

```
./.terraform/terraform.tfstate
./state/notes.tfstate
```

Το state γράφτηκε στο `state/notes.tfstate` και ο φάκελος `state` φτιάχτηκε αυτόματα. Το `.terraform/terraform.tfstate` δεν είναι state των resources σου, παρά το όνομα. Εκεί το `init` σημειώνει ποιο backend ρύθμισε.

TERMINAL

```
cat .terraform/terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "version": 3,
  "terraform_version": "1.16.4",
  "backend": {
    "type": "local",
    "config": {
      "path": "state/notes.tfstate",
      "workspace_dir": null
    },
    "hash": 2432518139
  }
}
```

Γι' αυτό κάθε αλλαγή στο block `backend` θέλει νέο `terraform init`. Άλλαξε το path και τρέξε plan χωρίς init.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }

  backend "local" {
    path = "state/team.tfstate"
  }
}

resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Σημειώσεις της ομάδας\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Backend initialization required, please run "terraform init"
│
│ Reason: Backend configuration block has changed
…
```

Το Terraform είδε ότι το block διαφέρει από αυτό που κατέγραψε το `init` και σταμάτησε χωρίς να αγγίξει τίποτα. Το `init` θέλει τότε `-migrate-state` για να μεταφέρει το state στη νέα θέση ή `-reconfigure` για να το αφήσει πίσω. Τα δύο εξηγούνται στο κεφάλαιο 19. Εδώ γύρισε το path στο `state/notes.tfstate` και όλα δουλεύουν όπως πριν.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το local backend είναι η προεπιλογή όταν δεν υπάρχει block `backend` ή `cloud`. Αποθηκεύει το state στο `terraform.tfstate` του φακέλου εργασίας και κλειδώνει το αρχείο όσο τρέχει μια εντολή. Κάθε αλλαγή backend απαιτεί `terraform init`.

Μυστικά μέσα στο state

Ο provider `local` έχει και το resource `local_sensitive_file`. Δουλεύει όπως το `local_file`, αλλά το περιεχόμενό του είναι **sensitive** και το Terraform δεν το τυπώνει. Φτιάξε τρίτο φάκελο με ένα αρχείο ρυθμίσεων που περιέχει κωδικό.

TERMINAL

```
cd ..
mkdir secrets
cd secrets
```



main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_sensitive_file" "db" {
  filename = "db.conf"
  content  = "password = Kr1f0s-K0dik0s\n"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_sensitive_file.db will be created
  + resource "local_sensitive_file" "db" {
      + content              = (sensitive value)
      …
      + directory_permission = "0700"
      + file_permission      = "0700"
      + filename             = "db.conf"
      + id                   = (known after apply)
```

```
    }

Plan: 1 to add, 0 to change, 0 to destroy.
local_sensitive_file.db: Creating...
local_sensitive_file.db: Creation complete after 0s [id=f4be32b14d7d15cd671563218b3a012b4df33ecf]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Στο plan το `content` γράφει `(sensitive value)` και οι προεπιλεγμένες άδειες είναι `0700`, ώστε μόνο εσύ να διαβάζεις το αρχείο. Το ίδιο κρύβει και το `terraform state show`.

TERMINAL

```
terraform state show local_sensitive_file.db
```

ΕΞΟΔΟΣ

```
…
    content              = (sensitive value)
…
```

Τώρα ψάξε τον κωδικό μέσα στο ίδιο το state.

TERMINAL

```
grep password terraform.tfstate
terraform show -json | jq -r '.values.root_module.resources[0].values.content'
```

ΕΞΟΔΟΣ

```
            "content": "password = Kr1f0s-K0dik0s\n",
password = Kr1f0s-K0dik0s
```

Ο κωδικός είναι εκεί, σε απλό κείμενο. Το Terraform χρειάζεται την πραγματική τιμή για να τη συγκρίνει με τη configuration στο επόμενο plan. Το sensitive κρύβει μόνο την εμφάνιση στην οθόνη. Το `terraform show -json` δίνει την τιμή ολόκληρη, γιατί το διαβάζουν προγράμματα. Όποιος διαβάζει το state διαβάζει και κάθε μυστικό του, δηλαδή κωδικούς βάσεων, κλειδιά και tokens.

Γι’ αυτό το state δεν μπαίνει ποτέ σε Git. Ένα `.gitignore` το αποκλείει μαζί με τον φάκελο `.terraform`. Εδώ αποκλείει και το ίδιο το `db.conf`. Το `*.tfstate.*` πιάνει το backup και το lock info.

.gitignore

```
.terraform/
*.tfstate
*.tfstate.*
db.conf
```

TERMINAL

```
git init -q
git status --short
```

ΕΞΟΔΟΣ

```
?? .gitignore
?? .terraform.lock.hcl
?? main.tf
```

Στο Git πηγαίνουν μόνο η configuration, το `.gitignore` και το lock file των providers.

ΠΑΓΙΔΑ

Ένα state που μπήκε έστω και μία φορά σε Git μένει στο ιστορικό, ακόμη κι αν το σβήσεις στο επόμενο commit. Αν συμβεί, θεώρησε ότι κάθε μυστικό του διέρρευσε και άλλαξέ το.

Η σωστή θέση για ένα state με μυστικά είναι ένα remote backend με κρυπτογράφηση και έλεγχο πρόσβασης, όπως θα δεις στο κεφάλαιο 19. Το κεφάλαιο 24 δείχνει τρόπους να μη φτάσει καθόλου ένα μυστικό στο state.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Οι τιμές που είναι sensitive γράφονται στο state σε απλό κείμενο. Το `sensitive` κρύβει μόνο την εμφάνιση στο plan και στην έξοδο των εντολών. Το state προστατεύεται σαν μυστικό. Δεν μπαίνει σε Git και αποθηκεύεται σε backend με κρυπτογράφηση και περιορισμένη πρόσβαση.

Γιατί δεν αλλάζεις το state με το χέρι

Το state είναι JSON και τίποτα δεν σε εμποδίζει να το ανοίξεις σε editor. Το Terraform όμως πιστεύει ό,τι γράφει. Γύρνα στο `team` και κράτησε αντίγραφο. Μετά άλλαξε με `jq` το `content` του `rules` μέσα στο state, όπως θα το άλλαζε κάποιος με editor.

TERMINAL

```
cd ../team
cp terraform.tfstate saved.tfstate
jq '(.resources[] | select(.name == "rules")
  | .instances[0].attributes.content) = "Κανένας κανόνας\n"' \
  saved.tfstate > terraform.tfstate
```

Το `rules.txt` στον δίσκο είναι σωστό. Άλλαξε μόνο η μνήμη του Terraform.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.welcome: Refreshing state... [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
local_file.rules: Refreshing state... [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.rules must be replaced
-/+ resource "local_file" "rules" {
      ~ content              = <<-EOT # forces replacement
```

```
          - Κανένας κανόνας
          + Πρώτα plan, μετά apply.
        EOT
    …
    ~ id                   = "5709f6ccf9c96c112ca7b3b252f703113f21fbe1" -> (known after apply)
      # (3 unchanged attributes hidden)
  }

Plan: 1 to add, 0 to change, 1 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `id` στο state ταιριάζει ακόμη με το αρχείο, οπότε το refresh δεν βρήκε πρόβλημα και κράτησε το `content` που έγραψες. Το Terraform σύγκρινε τη configuration με αυτό και είδε μια διαφορά που δεν υπάρχει. Το plan προτείνει `-/+`, δηλαδή να σβήσει και να ξαναφτιάξει ένα αρχείο που είναι ήδη σωστό. Στο AWS το ίδιο λάθος μπορεί να σημαίνει αντικατάσταση ενός server ή μιας βάσης. Επανάφερε το αντίγραφο.

TERMINAL

```
cp saved.tfstate terraform.tfstate
rm saved.tfstate
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.welcome: Refreshing state... [id=e6009ceb1ca0f0ed6124f6a86cafa7f57c7aaa2c]
local_file.rules: Refreshing state... [id=5709f6ccf9c96c112ca7b3b252f703113f21fbe1]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Άλλοι κίνδυνοι είναι χειρότεροι. Ένα κόμμα που λείπει κάνει το αρχείο άκυρο JSON και καμία εντολή δεν δουλεύει. Μια αλλαγή όσο τρέχει apply χάνεται όταν εκείνο γράψει. Για τις αλλαγές που πράγματι χρειάζονται, όπως μετονομασία ενός resource ή αφαίρεσή του από τη διαχείριση χωρίς διαγραφή, υπάρχουν οι υποεντολές του `terraform state`, όπως οι `state mv` και `state rm`. Γράφουν έγκυρο state, κρατούν backup και σέβονται το lock. Τις βλέπεις στο κεφάλαιο 21.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το state το γράφει μόνο το Terraform. Το διαβάζεις με `terraform state list`, `terraform state show` και `terraform show`. Το αλλάζεις με apply ή με τις εντολές `terraform state` του κεφαλαίου 21. Ποτέ με editor.

Τα αρχεία του state σε μία σελίδα

Αρχείο	Τι κρατά	Σε Git
`terraform.tfstate`	Το state του local backend	Όχι
`terraform.tfstate.backup`	Το state πριν από την τελευταία εντολή που το άλλαξε	Όχι

Αρχείο	Τι κρατά	Σε Git
`.terraform.tfstate.lock.info`	Ποιος κρατά το lock, όσο τρέχει μια εντολή	Όχι
`.terraform/terraform.tfstate`	Ποιο backend ρύθμισε το `init`	Όχι
`.terraform.lock.hcl`	Τις εκδόσεις των providers, όχι state	Ναι

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch04`, εκτός αν η εκφώνηση λέει αλλιώς. Γράψε το `main.tf` με τον provider `local` 2.9.1, τρέξε `terraform init` και μετά τις εντολές που ζητά η εκφώνηση. Για το apply χρησιμοποίησε `-auto-approve`.

ΑΣΚΗΣΗ 4.1 Serial και lineage

Γράψε configuration με ένα `local_file` με όνομα `note`, που φτιάχνει το `note.txt` με περιεχόμενο `πρώτη εκδοχή\n`. Κάνε apply και κράτησε το lineage με `jq -r .lineage terraform.tfstate > lineage.txt`. Άλλαξε το περιεχόμενο σε `δεύτερη εκδοχή\n` και κάνε ξανά apply. Τύπωσε το `serial` και έλεγξε με `diff` ότι το `lineage` δεν άλλαξε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
4
ίδιο lineage
```

ΑΣΚΗΣΗ 4.2 Λίστα και JSON

Γράψε configuration με τρία resources `local_file` με ονόματα `a`, `b` και `c`, που φτιάχνουν τα `a.txt`, `b.txt` και `c.txt`. Κάνε apply. Τύπωσε τις διευθύνσεις με `terraform state list`. Μετά βγάλε μόνο τα ονόματα των αρχείων από το `terraform show -json` με `jq -r`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
a.txt
b.txt
c.txt
```

ΑΣΚΗΣΗ 4.3 Ένα block που έφυγε

Ξεκίνα από αντίγραφο του φακέλου της προηγούμενης άσκησης, χωρίς το state του. Κάνε apply. Μετά σβήσε από το `main.tf` το block του `c` και τρέξε plan. Γιατί το Terraform ξέρει ότι πρέπει να σβήσει κάτι που δεν αναφέρεται πουθενά στη configuration;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # local_file.c will be destroyed
  # (because local_file.c is not in configuration)
…
Plan: 0 to add, 0 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 4.4 Αρχείο που χάθηκε

Γράψε configuration με ένα `local_file` με όνομα `report` για το `report.txt`. Κάνε apply και σβήσε το `report.txt` με `rm`. Τρέξε plan με `-no-color` και κράτησε με `grep` μόνο τη γραμμή της περίληψης. Αμέσως μετά τύπωσε τη λίστα του state. Θυμάται ακόμη το state το resource;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Plan: 1 to add, 0 to change, 0 to destroy.
local_file.report
```

ΑΣΚΗΣΗ 4.5 Τι κρατά το backup

Γράψε configuration με ένα `local_file` με όνομα `status`, που φτιάχνει το `status.txt` με περιεχόμενο `ανοιχτό\n`. Κάνε apply. Άλλαξε το περιεχόμενο σε `κλειστό\n` και κάνε apply. Κάνε και τρίτο apply χωρίς καμία αλλαγή. Μετά τύπωσε με `jq -j` το `content` πρώτα από το backup και μετά από το state.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ανοιχτό
κλειστό
```

ΑΣΚΗΣΗ 4.6 State σε άλλο φάκελο

Γράψε configuration με local backend που κρατά το state στο `state/dev.tfstate` και με ένα `local_file` με όνομα `env`, που φτιάχνει το `env.txt` με περιεχόμενο `dev\n`. Κάνε init και apply. Άλλαξε το περιεχόμενο σε `development\n` και κάνε ξανά apply. Βρες με `find` όλα τα αρχεία με `tfstate` στο όνομα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
./.terraform/terraform.tfstate
./state/dev.tfstate
./state/dev.tfstate.backup
```

ΑΣΚΗΣΗ 4.7 Το μυστικό και το Git

Γράψε configuration με ένα `local_sensitive_file` με όνομα `token`, που φτιάχνει το `token.txt` με περιεχόμενο `token = demo-token-42\n`. Κάνε apply και βρες το token μέσα στο `terraform.tfstate` με `grep`. Μετά γράψε `.gitignore` ώστε σε ένα νέο Git repository στον φάκελο το `git status --short` να δείχνει μόνο το `main.tf`, το `.gitignore` και το lock file των providers.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
?? .gitignore
?? .terraform.lock.hcl
?? main.tf
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 4.1

Ποιος είναι ο κύριος σκοπός του state στο Terraform;

a. Να κρατά τα credentials με τα οποία οι providers μιλούν στα APIs.
b. Να αντιστοιχίζει τα resources της configuration σε πραγματικά αντικείμενα και να κρατά metadata γι' αυτά.
c. Να αντικαθιστά την configuration όταν τα αρχεία `.tf` λείπουν.
d. Να καταγράφει ποιος χρήστης έτρεξε κάθε apply.

ΕΡΩΤΗΣΗ 4.2

Μια configuration δεν έχει block `backend` ούτε block `cloud`. Πού αποθηκεύει το Terraform το state;

a. Στο `.terraform/terraform.tfstate`
b. Στο `terraform.tfstate` του φακέλου εργασίας
c. Στο HCP Terraform, στον λογαριασμό του χρήστη
d. Πουθενά. Το state υπάρχει μόνο στη μνήμη όσο τρέχει η εντολή.

ΕΡΩΤΗΣΗ 4.3

Ένα resource έχει attribute που ο provider χαρακτηρίζει sensitive και το state βρίσκεται στο local backend. Ποιες δύο προτάσεις ισχύουν; Επίλεξε δύο.

a. Το plan δείχνει `(sensitive value)` στη θέση της τιμής.
b. Το Terraform κρυπτογραφεί την τιμή μέσα στο `terraform.tfstate`.
c. Η τιμή γράφεται στο `terraform.tfstate` σε απλό κείμενο.
d. Η τιμή δεν γράφεται στο state, ώστε να μη διαρρεύσει.

ΕΡΩΤΗΣΗ 4.4

Σωστό ή λάθος; Το `terraform plan` γράφει στο `terraform.tfstate` τις τιμές που βρήκε κατά το refresh.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 4.5

Τρέχεις πολλά applies με αλλαγές στην ίδια configuration. Ποιο πεδίο του `terraform.tfstate` μένει ίδιο σε όλη τη ζωή του state;

a. Το `serial`
b. Το `lineage`
c. Το `resources`
d. Το `terraform_version`

ΕΡΩΤΗΣΗ 4.6

Θέλεις να δεις όλα τα attributes του `local_file.rules` όπως τα θυμάται το state. Ποια εντολή χρησιμοποιείς;

a. `terraform state list local_file.rules`
b. `terraform state show local_file.rules`
c. `terraform output local_file.rules`
d. `terraform show local_file.rules`



ΚΕΦΑΛΑΙΟ 5

Providers, εκδόσεις και το lock file

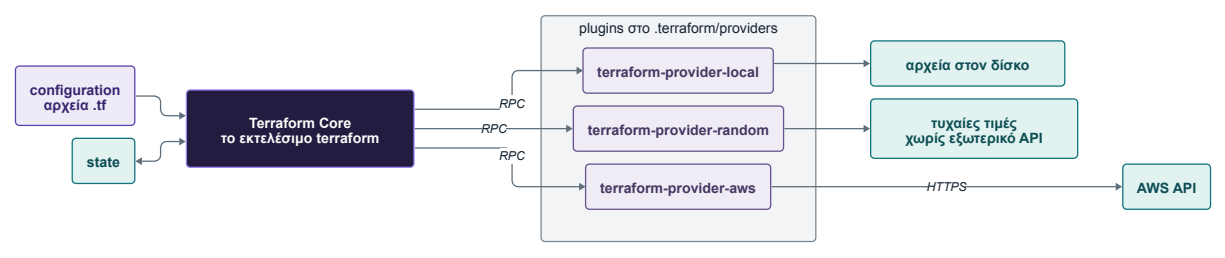
Το Terraform μόνο του δεν ξέρει να φτιάξει ούτε ένα αρχείο. Τη γνώση για κάθε πλατφόρμα την έχουν οι providers. Εδώ μαθαίνεις τι είναι, από πού έρχονται, πώς ζητάς έκδοση και πώς το lock file κρατά όλη την ομάδα στις ίδιες εκδόσεις.

Core και providers

Το εκτελέσιμο `terraform` λέγεται **Terraform Core**. Διαβάζει configuration και state, υπολογίζει το plan και αποφασίζει τη σειρά των ενεργειών. Δεν ξέρει όμως τι είναι ένα αρχείο ή ένα S3 bucket. Αυτό το ξέρουν οι providers.

Ένας **provider** είναι **plugin**, δηλαδή χωριστό εκτελέσιμο πρόγραμμα. Ο Core τον ξεκινά ως χωριστή διεργασία και του μιλά μέσω **RPC** (remote procedure call), ενός πρωτοκόλλου με το οποίο ένα πρόγραμμα καλεί λειτουργίες ενός άλλου. Ο provider κάνει τρία πράγματα.

1. Ορίζει τους τύπους resources και τα **data sources**. Το `local_file` υπάρχει επειδή το ορίζει ο `local`. Τα data sources διαβάζουν πληροφορίες χωρίς να φτιάχνουν τίποτα και τα βλέπεις στο κεφάλαιο 12.
2. Δίνει το schema κάθε τύπου, ποια arguments δέχεται και ποια attributes επιστρέφει. Από εκεί ήξερε το plan του κεφαλαίου 1 την προεπιλογή του `file_permission`.
3. Μεταφράζει κάθε ενέργεια του plan σε πραγματικές κλήσεις, στο σύστημα αρχείων για τον `local` και στο API του AWS για τον `aws`.



Σχήμα 5.1 · Ο Terraform Core διαβάζει configuration και state και μιλά με κάθε provider μέσω RPC. Κάθε provider είναι χωριστό εκτελέσιμο που ξέρει μία πλατφόρμα.

Έτσι ο Core μένει ίδιος και κάθε νέα πλατφόρμα χρειάζεται μόνο έναν νέο provider, που μπορεί να τον δημοσιεύσει οποιοσδήποτε. Κάθε provider έχει δικές του εκδόσεις, ανεξάρτητες από του Terraform.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Οι providers δεν είναι μέρος του εκτελέσιμου `terraform`. Κατεβαίνουν χωριστά με το `terraform init`, έχουν δικές τους εκδόσεις και μιλούν με τον Core μέσω RPC. Ο Core φτιάχνει το plan. Ο provider ξέρει τους τύπους resources και μιλά με το API.

Δύο providers σε μία configuration

Ο provider `random` φτιάχνει τυχαίες τιμές χωρίς να μιλά με κανένα εξωτερικό σύστημα. Η τιμή γράφεται στο state και μένει ίδια από apply σε apply. Βάλε τον δίπλα στον `local` για να φτιάξεις ένα αρχείο με τυχαίο όνομα.

TERMINAL

```
mkdir pets
cd pets
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "name" {}

resource "local_file" "pet" {
  filename = "${random_pet.name.id}.txt"
  content  = "Το κατοικίδιο λέγεται ${random_pet.name.id}\n"
}
```

Το `required_providers` έχει μία εγγραφή για κάθε provider. Το `random_pet` φτιάχνει ένα τυχαίο όνομα σαν όνομα κατοικίδιου. Τα άδεια άγκιστρα σημαίνουν ότι ισχύουν οι προεπιλογές. Το `${random_pet.name.id}`

μέσα στο string βάζει στη θέση του το `id` του `random_pet.name`, δηλαδή το όνομα. Αυτή η αναφορά σε άλλο resource λέγεται reference και εξηγείται στο κεφάλαιο 6.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/local versions matching "2.9.1"...
- Finding hashicorp/random versions matching "3.9.1"...
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Για κάθε provider το init γράφει μια γραμμή `Finding` με την έκδοση που ζητήσαμε και μετά τον εγκαθιστά. Πριν τρέξει τον κώδικα κάποιου άλλου στον υπολογιστή σου, ελέγχει την υπογραφή του. Και οι δύο είναι υπογεγραμμένοι από τη HashiCorp.

TERMINAL

```
cd .terraform/providers
find . -type f | sort
cd ../..
```

ΕΞΟΔΟΣ

```
./registry.terraform.io/hashicorp/local/2.9.1/darwin_arm64/LICENSE.txt
./registry.terraform.io/hashicorp/local/2.9.1/darwin_arm64/terraform-provider-local_v2.9.1_x5
./registry.terraform.io/hashicorp/random/3.9.1/darwin_arm64/LICENSE.txt
./registry.terraform.io/hashicorp/random/3.9.1/darwin_arm64/terraform-provider-random_v3.9.1_x5
```

Η διαδρομή έχει την ίδια δομή για κάθε provider. Registry, οργανισμός, τύπος, έκδοση και πλατφόρμα, δηλαδή λειτουργικό σύστημα και επεξεργαστής. Στη δική σου έξοδο η πλατφόρμα θα είναι η δική σου, για παράδειγμα `linux_amd64`. Το `terraform-provider-random_v3.9.1_x5` είναι το εκτελέσιμο του plugin. Το `_x5` λέει ότι μιλά την έκδοση 5 του πρωτοκόλλου των plugins.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.pet will be created
  + resource "local_file" "pet" {
      + content              = (known after apply)
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = (known after apply)
      + id                   = (known after apply)
    }

  # random_pet.name will be created
  + resource "random_pet" "name" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 2 to add, 0 to change, 0 to destroy.
random_pet.name: Creating...
random_pet.name: Creation complete after 0s [id=new-filly]
local_file.pet: Creating...
local_file.pet: Creation complete after 0s [id=3f7b775c0d79d0a5ca8fa1c8d75fa6f3bc1aaefe]

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

Το `random_pet.name` δείχνει τις προεπιλογές του, `length` δύο λέξεις με `separator` την παύλα. Το `filename` και το `content` του αρχείου γράφουν `(known after apply)`, γιατί εξαρτώνται από ένα όνομα που δεν υπάρχει ακόμη. Γι’ αυτό το Terraform έφτιαξε πρώτα το `random_pet.name` και μετά το αρχείο. Το όνομα φαίνεται σε αγκύλες στη γραμμή `Creation complete`. Στον δικό σου υπολογιστή θα είναι άλλο.

TERMINAL

```
ls
cat *.txt
```

ΕΞΟΔΟΣ

```
main.tf
new-filly.txt
terraform.tfstate
Το κατοικίδιο λέγεται new-filly
```

Δύο providers που δεν ξέρουν ο ένας τον άλλον δούλεψαν μαζί σε ένα plan. Τη σύνδεση την έκανε ο Core, που πήρε την τιμή από τον `random` και την έδωσε στον `local`. Τώρα και το `terraform version` αναφέρει τους providers του φακέλου.

TERMINAL

```
terraform version
```

ΕΞΟΔΟΣ

```
Terraform v1.16.4
on darwin_arm64
+ provider registry.terraform.io/hashicorp/local v2.9.1
+ provider registry.terraform.io/hashicorp/random v3.9.1
```

terraform providers

Η εντολή `terraform providers` δείχνει ποιους providers χρειάζεται η configuration και ποιος τους ζητά.

TERMINAL

```
terraform providers
```

ΕΞΟΔΟΣ

```
Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/local] 2.9.1
└── provider[registry.terraform.io/hashicorp/random] 3.9.1

Providers required by state:

    provider[registry.terraform.io/hashicorp/local]

    provider[registry.terraform.io/hashicorp/random]
```

Η πρώτη ενότητα είναι δέντρο. Η τελεία είναι ο τρέχων φάκελος, το root module. Κάτω του κρέμονται οι providers με το constraint τους. Όταν στο κεφάλαιο 15 προσθέσεις modules, κάθε module γίνεται κλαδί με τους δικούς του providers. Η δεύτερη ενότητα λέει ποιους providers χρειάζεται το state. Ένα resource του state θέλει τον provider του για να σβηστεί, ακόμη κι αν σβήσεις το block του.

Source address και Registry

Το `"hashicorp/random"` είναι η σύντομη μορφή της **source address**. Η πλήρης έχει τρία μέρη και το πρώτο μπορεί να λείπει.

ΕΞΟΔΟΣ

```
[hostname/]namespace/type
```

Το **hostname** είναι το registry που διανέμει τον provider. Όταν λείπει, ισχύει το `registry.terraform.io`, το δημόσιο **Terraform Registry**. Το **namespace** είναι ο οργανισμός που τον δημοσιεύει και το **type** το σύντομο όνομά του. Άρα το `hashicorp/random` είναι το ίδιο με το `registry.terraform.io/hashicorp/random`,

τη μορφή που είδες στη διαδρομή και στο `terraform providers`. Μια εταιρεία με δικό της private registry γράφει και το hostname, για παράδειγμα `registry.example.com/acme/internal`.

Το Registry δείχνει για κάθε provider τις εκδόσεις, τις πλατφόρμες και την τεκμηρίωση των resources του. Με ένα σήμα, το **tier**, λέει ποιος τον συντηρεί. Τα βασικά tiers είναι τρία.

- **Official.** Τους συντηρεί η HashiCorp. Σχεδόν όλοι έχουν namespace `hashicorp`, όπως ο `aws`, ο `local` και ο `random`.
- **Partner.** Τους γράφει και τους συντηρεί η εταιρεία της πλατφόρμας για το δικό της API, ως μέλος του προγράμματος συνεργατών της HashiCorp. Το namespace είναι της εταιρείας, όπως στον `cloudflare/cloudflare`.
- **Community.** Τους δημοσιεύουν μεμονωμένοι προγραμματιστές ή ομάδες στο δικό τους namespace, χωρίς έλεγχο από τη HashiCorp.

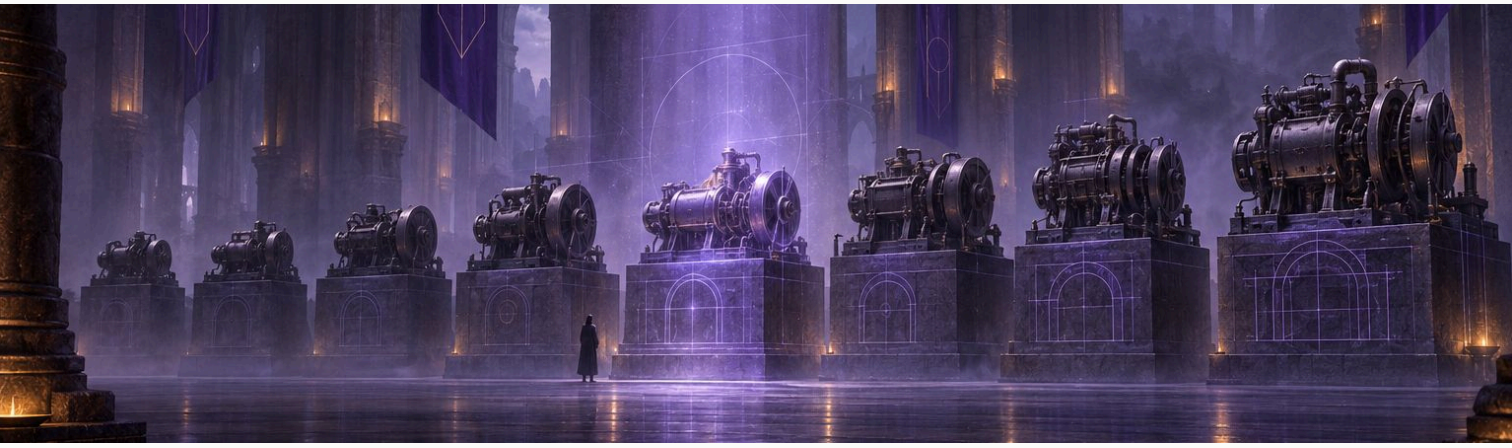
Ένας Official ή Partner provider που δεν συντηρείται πια παίρνει το σήμα **Archived**.

Το κλειδί μέσα στο `required_providers` είναι το **τοπικό όνομα** (local name) του provider. Έξω από εκεί η configuration αναφέρεται στους providers μόνο με αυτό. Για ένα resource `random_pet` το Terraform κρατά το πρόθεμα `random` πριν από την πρώτη κάτω παύλα και ψάχνει provider με αυτό το τοπικό όνομα. Γι' αυτό το τοπικό όνομα είναι σχεδόν πάντα ίδιο με τον τύπο. Διαφέρει όταν δύο providers έχουν τον ίδιο τύπο, για παράδειγμα ο `hashicorp/http` και ένας `acme/http`. Ο δεύτερος παίρνει άλλο όνομα, όπως `acme-http`. Κάθε resource του το δηλώνει τότε με `provider = acme-http`. Η άσκηση 5.5 δείχνει τι γίνεται αν το ξεχάσεις.

> **ΠΑΓΙΔΑ**
> Αν ένα resource θέλει provider που λείπει από το `required_providers`, το Terraform υποθέτει namespace `hashicorp` και γράφει `Finding latest version of hashicorp/random`. Παίρνει δηλαδή την πιο πρόσφατη έκδοση. Η υπόθεση για το namespace είναι σωστή μόνο για providers της HashiCorp. Δήλωνε πάντα `source` και `version`.

Version constraints

Το `version` είναι **version constraint**, ένας περιορισμός που λέει ποιες εκδόσεις γίνονται δεκτές. Οι εκδόσεις έχουν τη μορφή `MAJOR.MINOR.PATCH`. Το major αλλάζει όταν σπάει κάτι, το minor φέρνει νέες δυνατότητες και το patch διορθώσεις.



Τελεστής	Παράδειγμα	Δεκτές εκδόσεις
= ή κανένας	`"3.9.1"`	μόνο η 3.9.1
!=	`"!= 3.8.0"`	όλες εκτός από την 3.8.0
>	`"> 3.8.0"`	νεότερες από την 3.8.0
>=	`">= 3.8.0"`	η 3.8.0 και νεότερες
<	`"< 4.0.0"`	παλιότερες από την 4.0.0
<=	`"<= 3.9.1"`	η 3.9.1 και παλιότερες
~>	`"~> 3.8.0"`	η 3.8.0 και νεότερες της σειράς 3.8

Συνθήκες που χωρίζονται με κόμμα ισχύουν όλες μαζί. Το `">= 3.8.0, < 4.0.0"` δέχεται κάθε 3.x από την 3.8.0 και πάνω.

Ο `~>` λέγεται pessimistic constraint. Αφήνει να μεγαλώνει μόνο το τελευταίο ψηφίο που έγραψες. Το `~> 3.8.0` σημαίνει `>= 3.8.0, < 3.9.0` και σήμερα δίνει την 3.8.1. Το `~> 3.8` σημαίνει `>= 3.8, < 4.0` και δίνει την 3.9.1. Το `~> 3.0` δέχεται κάθε 3.x. Οι prereleases, όπως η `3.7.0-alpha1` του `random`, επιλέγονται μόνο όταν τις γράψεις ως ακριβή έκδοση.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Το `~> 1.2.0` δέχεται από την 1.2.0 έως πριν από την 1.3.0. Το `~> 1.2` δέχεται από την 1.2 έως πριν από την 2.0. Μέτρα πόσα ψηφία γράφτηκαν. Μόνο το τελευταίο μπορεί να μεγαλώσει.

Η έκδοση του ίδιου του Terraform

Το argument `required_version` στο block `terraform` περιορίζει με τον ίδιο τρόπο την έκδοση του CLI. Φαντάσου ένα project που κάποιος περιόρισε στη σειρά 1.5.

TERMINAL

```
cd ..
mkdir cli
cd cli
```

main.tf

```
terraform {
  required_version = "~> 1.5.0"

  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "note" {
  filename = "note.txt"
  content  = "Ποια έκδοση;\n"
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...


│ Error: Unsupported Terraform Core version
│
│   on main.tf line 2, in terraform:
│    2:   required_version = "~> 1.5.0"
│
│ This configuration does not support Terraform version 1.16.4. To proceed,
│ either choose another supported Terraform version or update this version
│ constraint. Version constraints are normally set for good reason, so
│ updating the constraint may lead to other errors or unexpected behavior.
```

Το σφάλμα `Unsupported Terraform Core version` δείχνει τη γραμμή του constraint και λέει ότι η configuration δεν υποστηρίζει την 1.16.4. Ο έλεγχος έγινε πριν από την εγκατάσταση των providers, οπότε ο `local` δεν κατέβηκε καν. Εγκαθιστάς έκδοση που ταιριάζει ή αλλάζεις το constraint, αφού βεβαιωθείς ότι ο κώδικας δουλεύει. Εδώ το αλλάζουμε σε κάθε 1.x από την 1.12 και μετά.

main.tf

```
terraform {
  required_version = ">= 1.12.0, < 2.0.0"

  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "note" {
  filename = "note.txt"
  content  = "Ποια έκδοση;\n"
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

Το `required_version` αφορά μόνο το CLI. Οι εκδόσεις των providers ορίζονται μόνο στο `required_providers`.

Ποια έκδοση διαλέγει το init

Ένα constraint όπως το `~> 3.0` δέχεται πολλές εκδόσεις. Δες ποια διαλέγει το init.

TERMINAL

```
cd ..
mkdir versions
cd versions
```

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "~> 3.0"
    }
  }
}

resource "random_pet" "name" {}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/random versions matching "~> 3.0"...
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Η γραμμή `Finding` γράφει τώρα το constraint. Το Terraform ζήτησε από το Registry τις εκδόσεις του `random` και διάλεξε τη νεότερη που ταιριάζει, την 3.9.1. Αν αύριο βγει μια 3.10.0, ένα init σε νέο φάκελο θα πάρει εκείνη.

Το lock file

TERMINAL

```
cat .terraform.lock.hcl
```

ΕΞΟΔΟΣ

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/random" {
  version     = "3.9.1"
  constraints = "~> 3.0"
  hashes = [
    "h1:PlW+UZ4EElQF3NQwf41KQwavFujab3Czc51zu9dyVM8=",
    "zh:05f4734c1f0be840b711b3eff259ebc5fca436784c728955b1678078466f48d7",
    "zh:0b91bf19371d012434eba1deeb6aab77158def9b39601dcbd94450b3974a2a26",
    "zh:0ee6eacd47ec00183d55d726a4b6c4ce951a199f944bf22f1aa58392ebdfa7a2",
    "zh:19388a4074b76a89a43a6c8328d7ae8ee2e7de3d346af51e80d3e6d3d12925f1",
    "zh:23e74d48c5e2ac2e823fd527f49fee9db37d32a1990c9e3bf126ead697b843eb",
    "zh:3cabf7fbd096c520064aae3aba61aba670af83ab91291a71fa1b1332929c2b7f",
    "zh:5c0a3b8af0be60be4eca12ddee385cfa8babc1ec8e98cdf9de2f2274c73eabfa",
    "zh:60b4f8a8ef18f52bf8e19215229dae408bee732825964092db7c989fd2de4097",
    "zh:7359015acfedcbd6366f2329c854cf8d3c8ca5cd0faa89d2d37db358d6eba6c5",
    "zh:78d5eefdd9e494defcb3c68d282b8f96630502cac21d1ea161f53cfe9bb483b3",
```

```
    "zh:7b38758402f0e13a1071162da28994023cd2ac676e54af350c9ffd8dfa73fa7b",
    "zh:7c7fbb8895eb75bb4de1f933e98553bd99c8d048c89a925ddba490aa5a67f7dc",
    "zh:8c2b8c6a7ccdec16b73e2fb9f3700ea097f58c592571e4c5de60c93d2301732c",
  ]
}
```

Το lock file είναι HCL και το συντηρεί το `terraform init`, όπως λένε τα σχόλια. Έχει ένα block `provider` για κάθε provider, με την πλήρη source address.

- Το `version` είναι η έκδοση που διάλεξε το init, η 3.9.1.
- Το `constraints` είναι το constraint τη στιγμή της επιλογής, το `~> 3.0`.
- Το `hashes` κρατά checksums, ώστε κάθε μελλοντική εγκατάσταση να είναι ακριβώς το ίδιο πακέτο.

Το `h1:` είναι hash των αρχείων του provider μετά την αποσυμπίεση. Το init το υπολογίζει από όσα εγκατέστησε, οπότε υπάρχει ένα, για τη δική σου πλατφόρμα. Τα `zh:` είναι τα SHA256 των αρχείων zip, ένα για το πακέτο κάθε πλατφόρμας. Έρχονται από τη λίστα checksums που υπογράφει ο εκδότης του provider.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/random v3.9.1

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Στο δεύτερο init δεν υπάρχει γραμμή `Finding`. Η γραμμή που αρχίζει με `Reusing` λέει ότι το init κράτησε την επιλογή του lock file. Ακόμη κι αν υπήρχε νεότερη 3.x, θα έμενε στην 3.9.1. Το constraint λέει ποιες εκδόσεις επιτρέπονται. Το lock file λέει ποια χρησιμοποιείται.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το `.terraform.lock.hcl` μπαίνει σε Git, ο φάκελος `.terraform` όχι. Έτσι όποιος τρέξει `terraform init` παίρνει την ίδια έκδοση κάθε provider, ελεγμένη με τα checksums. Το lock file καταγράφει μόνο providers, όχι modules.

Όταν αλλάζει το constraint

Θέλεις να μείνεις στη σειρά 3.8. Άλλαξε το constraint σε `~> 3.8.0`.

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "~> 3.8.0"
    }
  }
}

resource "random_pet" "name" {}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Inconsistent dependency lock file
│
│ The following dependency selections recorded in the lock file are
│ inconsistent with the current configuration:
│   - provider registry.terraform.io/hashicorp/random: locked version selection 3.9.1
doesn't match the updated version constraints "~> 3.8.0"
│
│ To update the locked dependency selections to match a changed
│ configuration, run:
│   terraform init -upgrade
╵
```

Το lock file λέει 3.9.1 και το νέο constraint δεν τη δέχεται. Το `Inconsistent dependency lock file` το λέει και δίνει τη λύση, `terraform init -upgrade`. Ένα σκέτο init δεν αρκεί.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
╷
│ Error: Failed to query available provider packages
│
│ Could not retrieve the list of available versions for provider
│ hashicorp/random: locked provider registry.terraform.io/hashicorp/random
│ 3.9.1 does not match configured version constraint ~> 3.8.0; must use
│ terraform init -upgrade to allow selection of new versions
│
│ To see which modules are currently depending on hashicorp/random and what
│ versions are specified, run the following command:
│     terraform providers
╵
```

Το init προσπάθησε να κρατήσει την κλειδωμένη 3.9.1 και απέτυχε. Μια κλειδωμένη έκδοση το init δεν την αλλάζει μόνο του. Την αλλάζει μόνο με τη σημαία `-upgrade`.

TERMINAL

```
terraform init -upgrade
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/random versions matching "~> 3.8.0"...
- Installing hashicorp/random v3.8.1...
- Installed hashicorp/random v3.8.1 (signed by HashiCorp)

Terraform has made some changes to the provider dependency selections recorded
in the .terraform.lock.hcl file. Review those changes and commit them to your
version control system if they represent changes you intended to make.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Με το `-upgrade` το init αγνοεί τις επιλογές του lock file, διαλέγει τη νεότερη έκδοση που ταιριάζει με το constraint και ενημερώνει το lock file. Εδώ αυτή είναι η 3.8.1, παλιότερη από την 3.9.1. Παρά το όνομά της, η σημαία πηγαίνει και προς τα πίσω όταν το ζητά το constraint.

TERMINAL

```
grep -E "version|constraints" .terraform.lock.hcl
```

ΕΞΟΔΟΣ

```
  version     = "3.8.1"
  constraints = "~> 3.8.0"
```

Με το ίδιο `-upgrade` ανεβαίνεις σε νεότερη έκδοση μέσα στο constraint σου, όπως δείχνει η άσκηση 5.4. Το κεφάλαιο 16 δείχνει ότι ενημερώνει και τα modules.

ΠΑΓΙΔΑ

Μη σβήνεις το lock file για να ξεφύγεις από τέτοιο σφάλμα. Το init θα φτιάξει νέο, αλλά δεν θα ξέρεις τι άλλαξε. Τρέξε `terraform init -upgrade`, δες τη διαφορά στο Git και κάνε commit.

Lock file για πολλές πλατφόρμες

Το lock file έχει ένα `h1:` ανά provider, για την πλατφόρμα του init. Όταν μια συνάδελφος σε Linux τρέξει init με lock file από macOS, ο έλεγχος γίνεται με τα `zh:`. Το init της όμως προσθέτει το δικό της

`h1:` και αλλάζει το lock file. Η εντολή `terraform providers lock` το προλαβαίνει. Γράφει από ένα `h1:` για κάθε πλατφόρμα που της δίνεις.

TERMINAL

```
cd ../pets
terraform providers lock -platform=linux_amd64 -platform=darwin_arm64 -platform=windows_amd64
```

ΕΞΟΔΟΣ

```
- Fetching hashicorp/random 3.9.1 for linux_amd64...
- Retrieved hashicorp/random 3.9.1 for linux_amd64 (signed by HashiCorp)
- Fetching hashicorp/local 2.9.1 for linux_amd64...
- Retrieved hashicorp/local 2.9.1 for linux_amd64 (signed by HashiCorp)
- Fetching hashicorp/local 2.9.1 for darwin_arm64...
- Retrieved hashicorp/local 2.9.1 for darwin_arm64 (signed by HashiCorp)
- Fetching hashicorp/random 3.9.1 for darwin_arm64...
- Retrieved hashicorp/random 3.9.1 for darwin_arm64 (signed by HashiCorp)
- Fetching hashicorp/local 2.9.1 for windows_amd64...
- Retrieved hashicorp/local 2.9.1 for windows_amd64 (signed by HashiCorp)
- Fetching hashicorp/random 3.9.1 for windows_amd64...
- Retrieved hashicorp/random 3.9.1 for windows_amd64 (signed by HashiCorp)
…

Success! Terraform has updated the lock file.

Review the changes in .terraform.lock.hcl and then commit to your
version control system to retain the new checksums.
```

TERMINAL

```
grep -c "h1:" .terraform.lock.hcl
```

ΕΞΟΔΟΣ

```
6
```

Η εντολή κατέβασε το πακέτο κάθε provider για κάθε πλατφόρμα και έλεγξε την υπογραφή. Τώρα το lock file έχει έξι γραμμές `h1:`, τρεις για καθέναν από τους δύο providers. Τρέχεις την εντολή μία φορά με τις πλατφόρμες της ομάδας και του CI και κάνεις commit.

Το block provider

Ένας provider μπορεί να έχει ρυθμίσεις. Ο `aws` θέλει να ξέρει σε ποιο region θα δουλέψει. Οι ρυθμίσεις μπαίνουν στο block `provider`, με label το τοπικό όνομα.

HCL

```
provider "aws" {
  region = "eu-central-1"
}
```

Ο `local` και ο `random` δεν έχουν ρυθμίσεις. Όταν λείπει το block, το Terraform χρησιμοποιεί κενή ρύθμιση. Μπορείς όμως να το γράψεις άδειο για να δεις τη μηχανική.



Ένα block `provider` χωρίς `alias` είναι η **προεπιλεγμένη ρύθμιση** (default configuration). Για δεύτερη ρύθμιση του ίδιου provider, για παράδειγμα τον `aws` σε δεύτερο region, γράφεις δεύτερο block με `alias`. Ένα resource τη διαλέγει με το argument `provider`, γραμμένο ως `<τοπικό όνομα>.<alias>`.

TERMINAL

```
cd ..
mkdir aliases
cd aliases
```

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

provider "random" {}

provider "random" {
  alias = "second"
}

resource "random_pet" "first" {}

resource "random_pet" "other" {
  provider = random.second
}
```

Το πρώτο block είναι η προεπιλεγμένη ρύθμιση. Θα μπορούσε να λείπει, αλλά έτσι οι δύο ρυθμίσεις φαίνονται δίπλα δίπλα. Το `random_pet.first` δεν ζητά τίποτα και παίρνει την προεπιλεγμένη. Το `random_pet.other` ζητά τη δεύτερη με `provider = random.second`, χωρίς εισαγωγικά, γιατί είναι αναφορά και όχι string.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Finding hashicorp/random versions matching "3.9.1"...
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)
…
```

Ο `random` κατέβηκε μία φορά. Οι δύο ρυθμίσεις μοιράζονται το ίδιο εκτελέσιμο.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.other: Creating...
random_pet.first: Creating...
random_pet.other: Creation complete after 0s [id=resolved-moth]
random_pet.first: Creation complete after 0s [id=keen-filly]
…
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

TERMINAL

```
grep '"provider"' terraform.tfstate
```

ΕΞΟΔΟΣ

```
      "provider": "provider[\"registry.terraform.io/hashicorp/random\"]",
      "provider": "provider[\"registry.terraform.io/hashicorp/random\"].second",
```

Το state σημειώνει για κάθε resource ποια ρύθμιση το έφτιαξε. Το δεύτερο έχει το `.second` στο τέλος. Οι ανάποδες κάθετοι υπάρχουν επειδή το state είναι JSON. Χωρίς τη ρύθμιση `second` το Terraform δεν μπορεί ούτε να σβήσει το `random_pet.other`. Το πρακτικό παράδειγμα με δύο regions του AWS έρχεται στο κεφάλαιο 17.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Ένα resource χωρίς argument `provider` παίρνει την προεπιλεγμένη ρύθμιση, το block χωρίς `alias`. Η εξέταση δείχνει συχνά δύο blocks `provider "aws"` και ρωτά σε ποιο region θα φτιαχτεί ένα resource.

Ένα workflow για κάθε πλατφόρμα

Τα εργαλεία IaC των ίδιων των παρόχων cloud είναι δεμένα στη δική τους πλατφόρμα. Το CloudFormation ξέρει μόνο το AWS. Στο Terraform, για να προσθέσεις τον `random` δίπλα στον `local`, χρειάστηκε μία εγγραφή στο `required_providers`. Η γλώσσα, το `init`, το `plan`, το `apply` και το state έμειναν

ίδια. Αυτό σημαίνει **service agnostic**. Ένα workflow για κάθε υπηρεσία που έχει API, από servers μέχρι εγγραφές DNS και repositories στο GitHub.

- **Multi cloud** είναι υποδομή σε περισσότερους από έναν παρόχους cloud, για παράδειγμα AWS και Azure. Μια ομάδα το κάνει για να διαλέγει την καλύτερη υπηρεσία κάθε παρόχου, για να αντέχει βλάβη ενός παρόχου ή για να μην εξαρτάται από έναν.
- **Hybrid cloud** είναι ο συνδυασμός δημόσιου cloud με δικό σου datacenter, για παράδειγμα VMware vSphere μαζί με AWS. Και τα δύο έχουν provider.

Μια configuration μπορεί να φτιάχνει server με τον `hashicorp/aws`, εγγραφή DNS στο Cloudflare με τον `cloudflare/cloudflare` και repository με τον `integrations/github`. Η διεύθυνση του server περνά στην εγγραφή DNS με reference, όπως το όνομα του `random_pet` πέρασε στο αρχείο. Ο Core βλέπει όλα τα resources σε ένα plan και ξέρει ότι η εγγραφή περιμένει τον server.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Το Terraform δεν κάνει τον κώδικα φορητό από cloud σε cloud. Ένα resource του AWS δεν γίνεται αυτόματα virtual machine στο Azure. Κοινά είναι η γλώσσα, το workflow και το state, όχι οι τύποι resources.

Ακριβής έκδοση ή `~>`

Το βιβλίο γράφει παντού ακριβή έκδοση, ώστε η έξοδος που τυπώνεται να είναι ίδια με τη δική σου. Μια νέα έκδοση provider μπορεί να αλλάξει ένα attribute ή ένα μήνυμα.

Σε πραγματικό project την ακριβή έκδοση την καρφώνει ήδη το lock file. Το constraint λέει μέχρι πού επιτρέπεται να ανέβεις όταν το αποφασίσεις.

- Στο root module, όπου τρέχεις `plan` και `apply`, ταιριάζει το `~>`, για παράδειγμα `~> 3.9`. Ανεβαίνεις με ένα `init -upgrade` που ελέγχεις, χωρίς να περάσεις κατά λάθος σε νέα major έκδοση.
- Σε ένα module που το χρησιμοποιούν άλλες configurations, θέμα των κεφαλαίων 15 και 16, γράφεις μόνο την ελάχιστη έκδοση με `>=`. Μια configuration χρησιμοποιεί μία έκδοση από κάθε provider, που πρέπει να ικανοποιεί τα constraints όλων των modules. Αν δύο modules καρφώνουν διαφορετική ακριβή έκδοση, το init αποτυγχάνει.

Ένα module δηλώνει επίσης μόνο την ελάχιστη έκδοση του Terraform, για παράδειγμα `required_version = ">= 1.12"`.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch05`, εκτός αν η εκφώνηση λέει διαφορετικά. Χρησιμοποίησε τις εκδόσεις του βιβλίου, εκτός αν η άσκηση ζητά άλλο constraint.

ΑΣΚΗΣΗ 5.1 Πλήρης διεύθυνση

Γράψε configuration που ζητά τον `local` με τη σύντομη source address και τον `random` με την πλήρη, που ξεκινά από το hostname. Πρόσθεσε ένα `random_pet` και ένα `local_file` που γράφει το όνομά του στο `pet.txt`. Τρέξε init και `terraform providers`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/local] 2.9.1
└── provider[registry.terraform.io/hashicorp/random] 3.9.1
```

ΑΣΚΗΣΗ 5.2 Δύο συνθήκες

Γράψε configuration μόνο με τον `random`, με constraint `">= 3.0.0, < 3.9.0"` και ένα `random_pet`. Πριν τρέξεις init, μάντεψε ποια έκδοση θα εγκαταστήσει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
- Finding hashicorp/random versions matching ">= 3.0.0, < 3.9.0"...
…
- Installed hashicorp/random v3.8.1 (signed by HashiCorp)
…
```

ΑΣΚΗΣΗ 5.3 Κάθε 1.x από την 1.12

Γράψε configuration με ένα `local_file` και `required_version` που δέχεται κάθε 1.x από την 1.12 και μετά, αλλά όχι την 2.0. Χρησιμοποίησε μόνο τον τελεστή `~>` και τρέξε init.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Terraform has been successfully initialized!
…
```

ΑΣΚΗΣΗ 5.4 Το lock file κρατά την έκδοση

Γράψε configuration μόνο με τον `random` στην ακριβή έκδοση `3.8.1` και ένα `random_pet`. Τρέξε init. Άλλαξε το constraint σε `~> 3.8` και τρέξε ξανά init. Ποια έκδοση έμεινε; Τέλος τρέξε `terraform init -upgrade` και δείξε τις γραμμές `version` και `constraints` του lock file.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  version     = "3.9.1"
  constraints = "~> 3.8"
```

ΑΣΚΗΣΗ 5.5 Τοπικό όνομα

Γράψε configuration που δηλώνει τον `hashicorp/random` με τοπικό όνομα `rnd` και έχει ένα `random_pet`. Τρέξε init και `terraform validate`. Διόρθωσε τη configuration ώστε το `validate` να μη βγάζει προειδοποίηση, κρατώντας το `rnd`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Success! The configuration is valid.
```

ΑΣΚΗΣΗ 5.6 Το alias που χάθηκε

Ξεκίνα από αντίγραφο του φακέλου `aliases`. Σβήσε το block `provider` με `alias = "second"`, άφησε το `random_pet.other` όπως είναι και τρέξε init και plan. Τι λέει το σφάλμα;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Provider configuration not present
╷
│ provider["registry.terraform.io/hashicorp/random"].second is required, but
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 5.1

Ένα module δηλώνει `version = "~> 1.2.0"` για έναν provider. Ποιες εκδόσεις δέχεται;

a. Μόνο την 1.2.0
b. Από την 1.2.0 έως πριν από την 2.0.0
c. Από την 1.2.0 έως πριν από την 1.3.0
d. Από την 1.2.0 και πάνω, χωρίς άνω όριο

ΕΡΩΤΗΣΗ 5.2

Μια configuration γράφει `source = "hashicorp/aws"`. Ποια είναι η πλήρης source address;

a. `hashicorp.com/aws`
b. `registry.terraform.io/hashicorp/aws`
c. `github.com/hashicorp/terraform-provider-aws`
d. `releases.hashicorp.com/hashicorp/aws`

ΕΡΩΤΗΣΗ 5.3

Μια συνάδελφος άλλαξε το constraint του `random` από `~> 3.0` σε `~> 3.8.0`. Το lock file γράφει 3.9.1 και το `terraform plan` αποτυγχάνει με `Inconsistent dependency lock file`. Τι πρέπει να τρέξει;

a. `terraform init`
b. `terraform init -upgrade`
c. Να σβήσει το `.terraform.lock.hcl` και να τρέξει `terraform plan`
d. `terraform validate`

ΕΡΩΤΗΣΗ 5.4

Ποιες δύο προτάσεις για τους providers είναι σωστές; Επίλεξε δύο.

a. Είναι χωριστά εκτελέσιμα που ο Terraform Core ξεκινά και με τα οποία μιλά μέσω RPC.
b. Είναι ενσωματωμένοι στο εκτελέσιμο `terraform` και αναβαθμίζονται μαζί του.
c. Κατεβαίνουν με το `terraform init` στον φάκελο `.terraform/providers`.
d. Η έκδοσή τους ορίζεται με το argument `required_version`.

ΕΡΩΤΗΣΗ 5.5

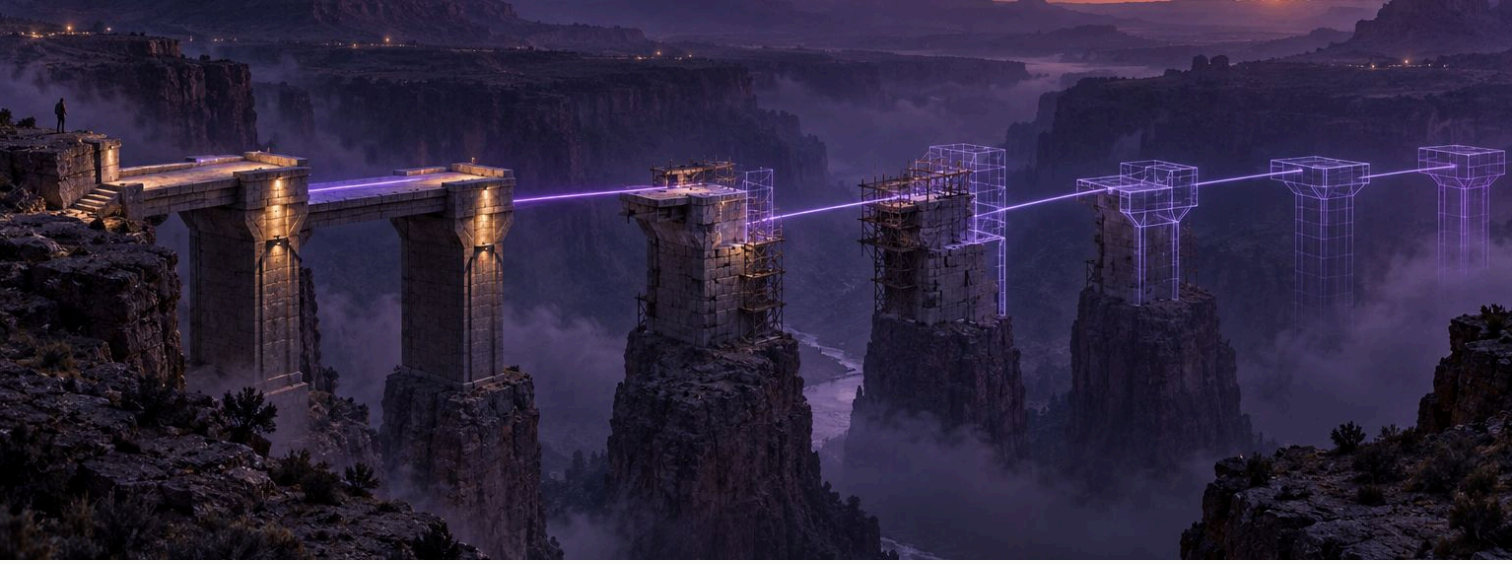
Σωστό ή λάθος; Μια configuration έχει δύο blocks `provider "aws"`, ένα χωρίς `alias` και ένα με `alias = "west"`. Ένα `aws_s3_bucket` χωρίς argument `provider` φτιάχνεται με τη ρύθμιση χωρίς `alias`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 5.6

Μια εταιρεία έχει servers σε δικό της datacenter, εφαρμογές στο AWS και στο Azure και DNS σε εξωτερική υπηρεσία. Ποιο είναι το βασικό όφελος αν τα διαχειρίζεται όλα με Terraform;

a. Το Terraform μεταφράζει αυτόματα τα resources του AWS σε ισοδύναμα του Azure.
b. Δεν χρειάζονται πια credentials για κάθε πλατφόρμα.
c. Κάθε πλατφόρμα κρατά δικό της εργαλείο και το Terraform ενώνει μόνο τα state.
d. Μία γλώσσα και ένα workflow για όλες τις πλατφόρμες, με references ανάμεσα σε resources διαφορετικών providers.



ΚΕΦΑΛΑΙΟ 6

Resources, references και dependencies

Μέχρι τώρα κάθε resource ζούσε μόνο του. Στην πράξη τα resources δένονται μεταξύ τους. Ένα όνομα που παράγει το ένα γίνεται argument του άλλου. Σε αυτό το κεφάλαιο μαθαίνεις να διαβάζεις τιμές από ένα resource, βλέπεις πώς το Terraform βγάζει από αυτές τη σειρά δημιουργίας και διαγραφής και πότε χρειάζεται να του πεις ρητά τι περιμένει τι.

Η διεύθυνση ενός resource

Από το κεφάλαιο 1 ξέρεις ότι ένα block `resource` έχει δύο labels, τον τύπο και το όνομα. Μαζί σχηματίζουν τη **διεύθυνση** του resource (resource address), για παράδειγμα `local_file.hello`. Η διεύθυνση είναι μοναδική μέσα στη configuration. Μπορείς να έχεις `local_file.hello` και `random_pet.hello`, γιατί οι τύποι διαφέρουν. Δεν μπορείς να έχεις δύο blocks `local_file` με όνομα `hello`.

Τη διεύθυνση την έδωσες ήδη στο `-replace` του κεφαλαίου 3 και στο `terraform state show` του κεφαλαίου 4. Αργότερα θα δεις και μακρύτερες. Ένα resource με `count` ή `for_each` παίρνει index ή key, όπως `local_file.page[0]` ή `local_file.page["home"]` (κεφάλαιο 11). Ένα resource μέσα σε module παίρνει πρόθεμα, όπως `module.site.local_file.page` (κεφάλαιο 15). Η βάση μένει πάντα `τύπος.όνομα`.

Φτιάξε τον πρώτο φάκελο του κεφαλαίου.

TERMINAL

```
mkdir pets
cd pets
```

Η configuration χρησιμοποιεί τους δύο providers που γνώρισες στο κεφάλαιο 5, τον `local` και τον `random`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "server" {
  prefix = "web"
}
```

Το `random_pet` φτιάχνει ένα τυχαίο όνομα από λέξεις, για παράδειγμα ένα επίθετο και ένα ζώο. Χρησιμεύει όταν ένα όνομα πρέπει να είναι μοναδικό, όπως το όνομα ενός S3 bucket. Το `prefix` μπαίνει μπροστά στο όνομα.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # random_pet.server will be created
  + resource "random_pet" "server" {
      + id        = (known after apply)
      + length    = 2
      + prefix    = "web"
      + separator = "-"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
random_pet.server: Creating...
random_pet.server: Creation complete after 0s [id=web-solid-labrador]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Στο plan το `prefix` έχει την τιμή που γράψαμε. Τα `length` και `separator` δεν τα γράψαμε. Είναι προεπιλογές του provider, δύο λέξεις ενωμένες με `-`. Το `id` γράφει `(known after apply)`. Το όνομα δεν υπάρχει πριν τρέξει ο provider, οπότε το plan δεν μπορεί να το ξέρει. Μετά το apply η γραμμή `Creation complete` δείχνει το όνομα σε αγκύλες. Το δικό σου θα είναι άλλο.

Arguments και attributes

Τα **arguments** είναι όσα γράφεις μέσα στο block. Εδώ μόνο το `prefix`. Τα **attributes** είναι όλες οι τιμές που εκθέτει το resource αφού φτιαχτεί, δηλαδή τα arguments, οι προεπιλογές και όσα υπολογίζει ο provider. Ποια attributes έχει κάθε τύπος το ορίζει ο provider. Τα βλέπεις με `terraform state show`.

TERMINAL

```
terraform state show random_pet.server
```

ΕΞΟΔΟΣ

```
# random_pet.server:
resource "random_pet" "server" {
    id        = "web-solid-labrador"
    length    = 2
    prefix    = "web"
    separator = "-"
}
```

Τα `length`, `prefix` και `separator` είναι arguments με την τιμή τους. Το `id` το υπολόγισε ο provider και είναι το ίδιο το όνομα. Στο `local_file` είδες περισσότερα υπολογισμένα attributes, όπως το `content_md5`. Ένα τέτοιο attribute είναι γνωστό μόνο μετά το apply. Μέχρι τότε το plan γράφει `(known after apply)`.

Η τυχαία τιμή παράγεται μία φορά και μένει στο state. Αλλάζει μόνο αν αντικατασταθεί το ίδιο το resource, για παράδειγμα αν αλλάξεις το `prefix`.

Αναφορές σε attributes

Ένα resource διαβάζει attributes άλλου resource με μια **αναφορά** (reference). Η αναφορά γράφεται `τύπος.όνομα.attribute`, για παράδειγμα `random_pet.server.id`. Μέσα σε string τη βάζεις σε `${...}`, όπως είδες στο κεφάλαιο 2. Αν η τιμή ενός argument είναι μόνο η αναφορά, τη γράφεις σκέτη, χωρίς εισαγωγικά. Στο απόσπασμα το argument `source`, που ορίζει ποιο αρχείο θα αντιγραφεί, παίρνει το `filename` ενός άλλου `local_file`.

HCL

```
resource "local_file" "copy" {
  filename = "copy.txt"
  source   = local_file.original.filename
}
```

Ένα συνηθισμένο λάθος είναι να μαντέψεις το όνομα του attribute. Γράψε ένα δεύτερο αρχείο που υποθέτει ότι το όνομα βρίσκεται στο `name`.

name.tf

```
resource "local_file" "name" {
  filename = "name.txt"
  content  = "${random_pet.server.name}\n"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Unsupported attribute
│
│   on name.tf line 3, in resource "local_file" "name":
│    3:   content  = "${random_pet.server.name}\n"
│
│ This object has no argument, nested block, or exported attribute named
│ "name".
╵
```

Το `terraform validate` ξέρει το schema του provider και βρίσκει ότι το `random_pet` δεν έχει `name`. Το σωστό attribute είναι το `id`, όπως έδειξε το `terraform state show`.

name.tf

```
resource "local_file" "name" {
  filename = "name.txt"
  content  = "${random_pet.server.id}\n"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
random_pet.server: Refreshing state... [id=web-solid-labrador]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.name will be created
  + resource "local_file" "name" {
      + content              = <<-EOT
            web-solid-labrador
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "name.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
local_file.name: Creating...
local_file.name: Creation complete after 0s [id=54ebc9a4e67de453cb9b51db82ebac431ec4631a]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

TERMINAL

```
cat name.txt
```

ΕΞΟΔΟΣ

```
web-solid-labrador
```

Το `random_pet.server` υπήρχε ήδη στο state, οπότε το Terraform ήξερε το `id` του πριν από το apply. Γι' αυτό το plan δείχνει το `content` με το πραγματικό όνομα και όχι `(known after apply)`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση δείχνει αναφορές όπως `aws_instance.web.public_ip` και ρωτά τι σημαίνει κάθε κομμάτι. Το πρώτο είναι ο τύπος, το δεύτερο το όνομα που έδωσες στο block και το τρίτο ένα attribute του resource.

Implicit dependencies

Μια αναφορά κάνει δύο δουλειές. Φέρνει την τιμή και δημιουργεί μια **εξάρτηση** (dependency). Αν το resource B διαβάζει attribute του A, το B δεν μπορεί να φτιαχτεί πριν υπάρξει το A. Το Terraform το καταλαβαίνει μόνο του από την αναφορά. Αυτή λέγεται **implicit dependency**, γιατί δεν τη δηλώνεις ξεχωριστά.

Το επόμενο παράδειγμα έχει αλυσίδα τριών επιπέδων. Ένα `random_pet` δίνει όνομα. Ένα `local_file` γράφει ένα αρχείο ρυθμίσεων με αυτό το όνομα. Ένα δεύτερο `local_file` γράφει μια σύνοψη που αναφέρει το πρώτο αρχείο.

TERMINAL

```
mkdir ../chain
cd ../chain
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "server" {}

resource "random_integer" "port" {
  min = 8000
  max = 8999
}

resource "local_file" "config" {
```

```
  filename = "${random_pet.server.id}.conf"
  content  = "name = ${random_pet.server.id}\nport = ${random_integer.port.result}\n"
}

resource "local_file" "summary" {
  filename = "summary.txt"
  content  = "Η ρύθμιση γράφτηκε στο ${local_file.config.filename}\n"
}
```

Διάβασε τα blocks ένα ένα.

- Το `random_pet.server` δεν έχει arguments. Το `{}` σημαίνει ότι όλα παίρνουν την προεπιλογή τους.
- Το `random_integer.port` διαλέγει έναν τυχαίο ακέραιο από το `min` έως το `max`. Τον εκθέτει ως αριθμό στο attribute `result`.
- Το `local_file.config` παίρνει το όνομα του αρχείου από το `random_pet.server.id` και γράφει μέσα το όνομα και τη θύρα. Έχει δύο αναφορές, άρα δύο εξαρτήσεις.
- Το `local_file.summary` αναφέρει το `local_file.config.filename`. Εξαρτάται από το `config` και μέσω αυτού από τα δύο τυχαία resources.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.config will be created
  + resource "local_file" "config" {
      + content              = (known after apply)
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = (known after apply)
      + id                   = (known after apply)
    }

  # local_file.summary will be created
  + resource "local_file" "summary" {
      + content              = (known after apply)
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "summary.txt"
      + id                   = (known after apply)
    }

  # random_integer.port will be created
  + resource "random_integer" "port" {
```

```
      + id     = (known after apply)
      + max    = 8999
      + min    = 8000
      + result = (known after apply)
    }

  # random_pet.server will be created
  + resource "random_pet" "server" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 4 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το plan αναφέρει τα resources με αλφαβητική σειρά, όχι με τη σειρά που θα φτιαχτούν. Κοίτα πού εμφανίζεται το `(known after apply)`. Στο `local_file.config` το `filename` και το `content` είναι άγνωστα, γιατί εξαρτώνται από τιμές που δεν υπάρχουν ακόμη. Στο `local_file.summary` το `filename` είναι σταθερό string και άρα γνωστό. Το `content` του όμως περιέχει το `filename` του `config` και είναι άγνωστο. Το άγνωστο περνά από κρίκο σε κρίκο της αλυσίδας.

TERMINAL

```
terraform apply -auto-approve
```

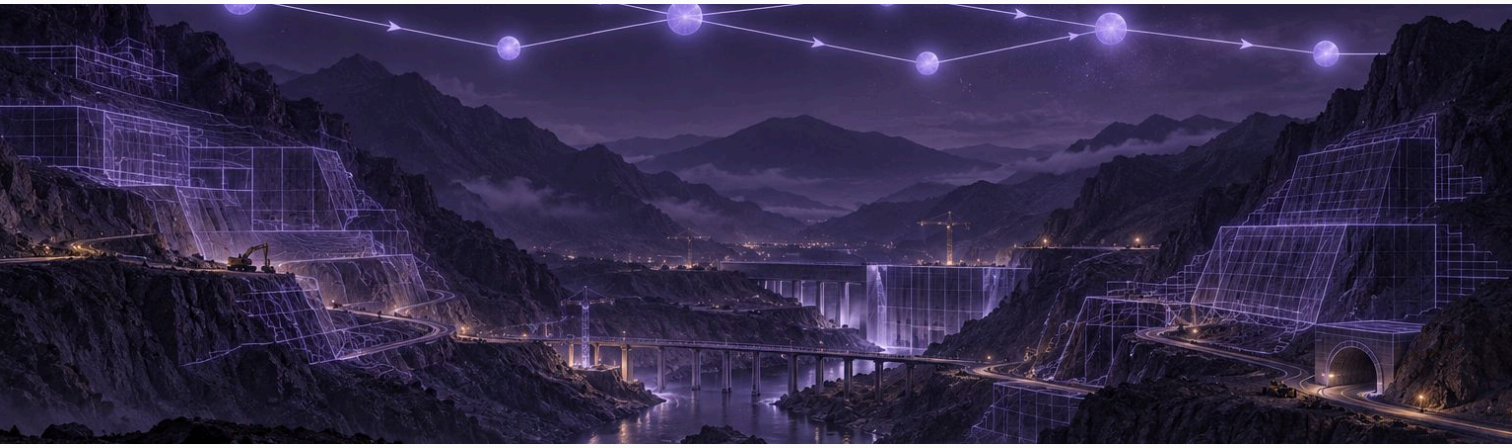
ΕΞΟΔΟΣ

```
…
random_pet.server: Creating...
random_integer.port: Creating...
random_pet.server: Creation complete after 0s [id=faithful-unicorn]
random_integer.port: Creation complete after 0s [id=8497]
local_file.config: Creating...
local_file.config: Creation complete after 0s [id=34994d55f0dc133df1f2612810604f50556e4dee]
local_file.summary: Creating...
local_file.summary: Creation complete after 0s [id=77b677ff38af545203a1466ba023147a740e8c04]
…
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

Η έξοδος του apply δείχνει τη σειρά. Το `random_pet.server` και το `random_integer.port` δεν εξαρτώνται από τίποτα, οπότε ξεκινούν αμέσως και χωρίς να περιμένει το ένα το άλλο. Η σειρά ανάμεσά τους αλλάζει από εκτέλεση σε εκτέλεση. Το `local_file.config` ξεκινά μόνο αφού τελειώσουν και τα δύο. Το `local_file.summary` ξεκινά τελευταίο.

TERMINAL

```
cat summary.txt
cat *.conf
```



ΕΞΟΔΟΣ

```
Η ρύθμιση γράφτηκε στο faithful-unicorn.conf
name = faithful-unicorn
port = 8497
```

ΚΡΑΤΑ ΑΥΤΟ

Μια αναφορά σε attribute άλλου resource είναι και εξάρτηση. Το Terraform φτιάχνει πρώτα αυτό που αναφέρεται και μετά αυτό που αναφέρει. Η σειρά των blocks στα αρχεία δεν παίζει κανέναν ρόλο.

Το graph των εξαρτήσεων

Πριν από κάθε plan και apply το Terraform χτίζει ένα **graph**. Κάθε resource είναι ένας κόμβος. Κάθε εξάρτηση είναι ένα βέλος από το resource που χρειάζεται κάτι προς αυτό που χρειάζεται. Το graph πρέπει να είναι κατευθυνόμενο και χωρίς κύκλους, ώστε να υπάρχει πάντα σειρά που σέβεται όλα τα βέλη. Το `terraform graph` το τυπώνει.

TERMINAL

```
terraform graph
```

ΕΞΟΔΟΣ

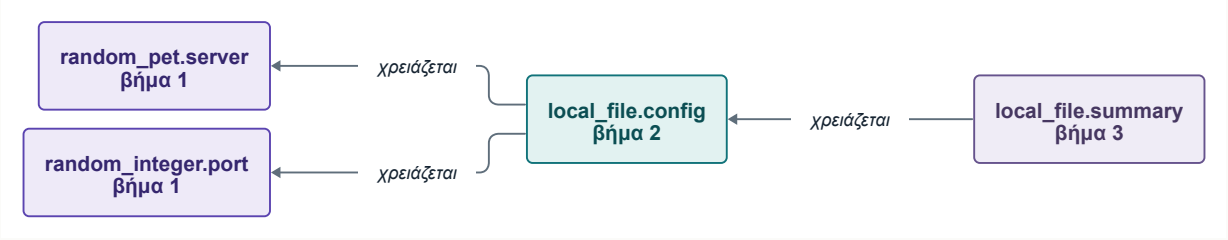
```
digraph G {
  rankdir = "RL";
  node [shape = rect, fontname = "sans-serif"];
  "local_file.config" [label="local_file.config"];
  "local_file.summary" [label="local_file.summary"];
  "random_integer.port" [label="random_integer.port"];
  "random_pet.server" [label="random_pet.server"];
  "local_file.config" -> "random_integer.port";
  "local_file.config" -> "random_pet.server";
  "local_file.summary" -> "local_file.config";
}
```

Η έξοδος είναι στη γλώσσα **DOT**, μια απλή γλώσσα περιγραφής γράφων. Οι γραμμές με `[label=...]` είναι οι κόμβοι, ένας για κάθε resource. Οι γραμμές με `->` είναι τα βέλη. Το `"local_file.config" ->`

`"random_pet.server"` διαβάζεται «το `config` χρειάζεται το `server`». Το `rankdir = "RL"` λέει στο πρόγραμμα που θα το σχεδιάσει να βάλει τα βέλη από δεξιά προς τα αριστερά. Το GraphViz κάνει το DOT εικόνα.

TERMINAL

```
terraform graph | dot -Tsvg > graph.svg
```



Σχήμα 6.1 · Το graph του φακέλου chain. Κάθε βέλος ξεκινά από ένα resource και δείχνει αυτό που χρειάζεται. Το apply δημιουργεί από αριστερά προς τα δεξιά και το destroy σβήνει από δεξιά προς τα αριστερά.

Το Terraform διασχίζει το graph ξεκινώντας από τους κόμβους που δεν χρειάζονται τίποτα. Μόλις τελειώσει ένας κόμβος, ελευθερώνονται όσοι περίμεναν μόνο αυτόν. Έτσι προκύπτουν τα τρία βήματα του σχήματος.

ΣΗΜΕΙΩΣΗ

Χωρίς επιλογές το `terraform graph` δείχνει μόνο τα resources. Με `-type=plan` δείχνει το πλήρες graph του plan, μαζί με κόμβους για τους providers. Το `-format=mermaid` του Terraform 1.16 βγάζει το graph σε μορφή Mermaid αντί για DOT. Είναι νεότερο από την ύλη της εξέτασης.

Η σειρά του destroy

Το destroy ακολουθεί το ίδιο graph ανάποδα. Ένα resource σβήνεται μόνο όταν δεν το χρειάζεται πια κανένα άλλο.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
local_file.summary: Destroying... [id=77b677ff38af545203a1466ba023147a740e8c04]
local_file.summary: Destruction complete after 0s
local_file.config: Destroying... [id=34994d55f0dc133df1f2612810604f50556e4dee]
local_file.config: Destruction complete after 0s
random_pet.server: Destroying... [id=faithful-unicorn]
random_integer.port: Destroying... [id=8497]
random_pet.server: Destruction complete after 0s
random_integer.port: Destruction complete after 0s
…
Destroy complete! Resources: 4 destroyed.
```

Πρώτο έφυγε το `local_file.summary`, που δεν το χρειαζόταν κανείς. Μετά το `local_file.config`. Τελευταία σβήστηκαν τα δύο τυχαία resources, πάλι παράλληλα. Σε ένα πραγματικό cloud η σειρά αυτή είναι αναγκαία. Ένα subnet δεν σβήνεται όσο υπάρχει server μέσα του.

ΚΡΑΤΑ ΑΥΤΟ

Create από τις εξαρτήσεις προς όσα εξαρτώνται από αυτές. Destroy με την αντίστροφη σειρά. Και τα δύο βγαίνουν από το ίδιο graph.

Parallelism

Όσα resources δεν συνδέονται με βέλος δεν έχουν σειρά μεταξύ τους. Το Terraform τα επεξεργάζεται παράλληλα, έως δέκα ταυτόχρονα με την προεπιλογή. Ο αριθμός αλλάζει με `-parallelism=N` στα `plan`, `apply` και `destroy`. Φτιάξε τρία ανεξάρτητα resources για να το δεις.

TERMINAL

```
mkdir ../parallel
cd ../parallel
```

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "name" {}

resource "random_integer" "port" {
  min = 8000
  max = 8999
}

resource "random_string" "code" {
  length  = 8
  special = false
}
```

Το `random_string` είναι ο τρίτος τυχαίος τύπος του κεφαλαίου. Φτιάχνει string με μήκος `length`. Με `special = false` δεν βάζει σύμβολα όπως `!` ή `#`, μόνο γράμματα και ψηφία. Με `upper = false` αφήνει έξω και τα κεφαλαία. Την τιμή την εκθέτει στο attribute `result`.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.name: Creating...
random_integer.port: Creating...
random_pet.name: Creation complete after 0s [id=lucky-polecat]
random_string.code: Creating...
random_integer.port: Creation complete after 0s [id=8984]
random_string.code: Creation complete after 0s [id=udqwOdDj]
…
```

Το `Creating...` ενός resource εμφανίζεται πριν από το `Creation complete` κάποιου άλλου. Το Terraform δεν περιμένει να τελειώσει το ένα για να ξεκινήσει το επόμενο. Τώρα σβήσε τα τρία resources και φτιάξ' τα ξανά με ένα μόνο ταυτόχρονο βήμα.

TERMINAL

```
terraform destroy -auto-approve
```

TERMINAL

```
terraform apply -auto-approve -parallelism=1
```

ΕΞΟΔΟΣ

```
…
random_integer.port: Creating...
random_integer.port: Creation complete after 0s [id=8327]
random_pet.name: Creating...
random_pet.name: Creation complete after 0s [id=enormous-sole]
random_string.code: Creating...
random_string.code: Creation complete after 0s [id=jmSqYekZ]
…
```

Τώρα κάθε `Creating...` ακολουθείται αμέσως από το δικό του `Creation complete`. Το Terraform δουλεύει ένα resource τη φορά. Η σειρά ανάμεσα στα τρία παραμένει τυχαία, γιατί το graph δεν ορίζει καμία.

Η προεπιλογή ταιριάζει σχεδόν πάντα. Μικρότερο νούμερο βοηθά όταν το API ενός παρόχου απορρίπτει πολλές ταυτόχρονες κλήσεις. Μεγαλύτερο βοηθά σε configurations με εκατοντάδες ανεξάρτητα resources, αν το API το αντέχει. Σε κάθε περίπτωση το `-parallelism` αλλάζει μόνο πόσα resources δουλεύουν μαζί, ποτέ τη σειρά που ορίζουν οι εξαρτήσεις.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η προεπιλογή του `-parallelism` είναι 10. Η εξέταση το ρωτά ευθέως ή ως μέρος ενός σεναρίου με όρια στο API του παρόχου.

Κρυφές εξαρτήσεις και depends_on

Το Terraform βλέπει μόνο τις εξαρτήσεις που περνούν από αναφορές. Δεν καταλαβαίνει τι σημαίνει ένα string. Στο επόμενο παράδειγμα το `local_file.backup` αντιγράφει με το argument `source` ένα αρχείο που φτιάχνει άλλο resource της ίδιας configuration.



TERMINAL

```
mkdir ../backup
cd ../backup
```

Αυτή τη φορά το block `terraform` μπαίνει σε δικό του αρχείο, το `versions.tf`. Το Terraform διαβάζει όλα τα αρχεία `.tf` του φακέλου μαζί (κεφάλαιο 2), οπότε το `main.tf` μένει μικρό όταν το αλλάζουμε.

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

main.tf

```
resource "random_pet" "server" {}

resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Ο server λέγεται ${random_pet.server.id}\n"
}

resource "local_file" "backup" {
  filename = "backup/notes.txt"
  source   = "notes.txt"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.server: Creating...
local_file.backup: Creating...
random_pet.server: Creation complete after 0s [id=rational-sturgeon]
local_file.notes: Creating...
local_file.notes: Creation complete after 0s [id=0d09173524d4af1293bf9834492efbe2af50f93d]
…
│ Error: Create local file error
…
│ +Original Error: open notes.txt: no such file or directory
…
```

Για το Terraform το `"notes.txt"` είναι απλό κείμενο. Δεν υπάρχει βέλος από το `backup` προς το `notes`, οπότε το `backup` δεν περίμενε τίποτα. Ξεκίνησε πριν γραφτεί το `notes.txt` και ο provider απέτυχε με `no such file or directory`. Το `local_file.notes` περίμενε πρώτα το όνομα από το `random_pet` και τελείωσε πολύ αργά για το `backup`. Η σειρά των γραμμών `Creating...` αλλάζει από εκτέλεση σε εκτέλεση, το σφάλμα όμως μένει.

Το `depends_on` δηλώνει μια εξάρτηση ρητά. Αυτή λέγεται **explicit dependency**, σε αντίθεση με την implicit που φτιάχνει μια αναφορά. Δέχεται λίστα από ολόκληρα resources. Είναι **meta-argument**, δηλαδή δεν ανήκει στο schema κάποιου provider. Το καταλαβαίνει το ίδιο το Terraform και γράφεται σε κάθε block `resource`. Σβήσε πρώτα ό,τι έμεινε, ώστε το επόμενο apply να ξεκινήσει από την αρχή.

TERMINAL

```
terraform destroy -auto-approve
```

main.tf

```
resource "random_pet" "server" {}

resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Ο server λέγεται ${random_pet.server.id}\n"
}

resource "local_file" "backup" {
  filename = "backup/notes.txt"
  source   = "notes.txt"

  depends_on = [local_file.notes]
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.server: Creating...
random_pet.server: Creation complete after 0s [id=literate-wahoo]
local_file.notes: Creating...
local_file.notes: Creation complete after 0s [id=85a4eb457506ac3971de65196c6254ab48fc89ad]
local_file.backup: Creating...
local_file.backup: Creation complete after 0s [id=85a4eb457506ac3971de65196c6254ab48fc89ad]
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

TERMINAL

```
cat backup/notes.txt
```

ΕΞΟΔΟΣ

```
Ο server λέγεται literate-wahoo
```

Τώρα η σειρά είναι `random_pet.server`, `local_file.notes`, `local_file.backup`. Το `depends_on` πρόσθεσε ένα βέλος στο graph, ίδιο με βέλος από αναφορά.

TERMINAL

```
terraform graph
```

ΕΞΟΔΟΣ

```
…
  "local_file.backup" -> "local_file.notes";
  "local_file.notes" -> "random_pet.server";
…
```

ΠΑΓΙΔΑ

Το `depends_on` είναι η εξαίρεση, όχι ο κανόνας. Όπου μπορείς να γράψεις αναφορά, γράψε αναφορά. Εδώ μπορούσες, γιατί το `local_file.notes.filename` είναι attribute. Η άσκηση 6.4 το δείχνει. Η αναφορά φέρνει την τιμή και την εξάρτηση μαζί. Ένα περιττό `depends_on` κρύβει τον λόγο της σειράς και κάνει τα plans πιο συντηρητικά. Κράτα το για εξαρτήσεις που δεν περνούν από καμία τιμή. Το κλασικό παράδειγμα είναι ένας server του AWS που χρειάζεται μια άδεια IAM την ώρα που τρέχει. Κανένα argument του server δεν αναφέρει την άδεια, αλλά χωρίς αυτήν το λογισμικό του αποτυγχάνει.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `depends_on` δέχεται λίστα με ολόκληρα resources ή modules, όπως `[local_file.notes]`. Ένα attribute, όπως `[local_file.notes.filename]`, είναι σφάλμα. Η εξέταση ρωτά πότε χρειάζεται το `depends_on`. Η απάντηση είναι όταν ένα resource χρειάζεται να υπάρχει πρώτα ένα άλλο, αλλά δεν χρησιμοποιεί καμία τιμή του σε κανένα argument. Αν η τιμή που χρειάζεσαι είναι attribute του άλλου resource, όπως το `filename` εδώ, η σωστή απάντηση είναι μια αναφορά.

terraform_data

Το `terraform_data` είναι resource ενσωματωμένο στο ίδιο το Terraform. Δεν φτιάχνει τίποτα έξω από το state και δεν χρειάζεται εγγραφή στο `required_providers`. Έχει δύο arguments.

- Το `input` δέχεται οποιαδήποτε τιμή. Το Terraform την αποθηκεύει και την εκθέτει στο attribute `output`. Μια αλλαγή στο `input` είναι update in place.
- Το `triggers_replace` δέχεται κι αυτό οποιαδήποτε τιμή. Όταν η τιμή αλλάξει, το resource αντικαθίσταται.

Χρησιμεύει για να κρατάς μια τιμή στο state και για να προκαλείς αντικαταστάσεις. Το κεφάλαιο 14 το συνδυάζει με το `replace_triggered_by`, που αντικαθιστά άλλα resources όταν αλλάζει αυτό. Υπάρχει από το Terraform 1.4. Παλαιότερες configurations χρησιμοποιούσαν για τον ίδιο σκοπό το `null_resource` του provider `null`.

TERMINAL

```
mkdir ../release
cd ../release
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

main.tf

```
resource "terraform_data" "release" {
  input            = "1.0.0"
  triggers_replace = "deploy-1"
}

resource "local_file" "release" {
  filename = "release.txt"
  content  = "Έκδοση ${terraform_data.release.output}\n"
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- terraform.io/builtin/terraform is built in to Terraform
…
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
```

Το `init` αναφέρει το `terraform.io/builtin/terraform` ως ενσωματωμένο και δεν κατεβάζει τίποτα για αυτό. Κατεβάζει μόνο τον `local`.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
terraform state show terraform_data.release
cat release.txt
```

ΕΞΟΔΟΣ

```
# terraform_data.release:
resource "terraform_data" "release" {
    id               = "026f6bc9-429b-0ee8-47ad-aaec4ef16506"
    input            = "1.0.0"
    output           = "1.0.0"
    triggers_replace = "deploy-1"
}
Έκδοση 1.0.0
```

Το `output` έχει την τιμή του `input`. Το `id` είναι ένα τυχαίο UUID που έδωσε το Terraform. Το `local_file.release` διάβασε το `output` με αναφορά, οπότε φτιάχτηκε μετά το `terraform_data`. Άλλαξε τώρα μόνο το `input`.

main.tf

```
resource "terraform_data" "release" {
  input            = "1.1.0"
  triggers_replace = "deploy-1"
}

resource "local_file" "release" {
  filename = "release.txt"
  content  = "Έκδοση ${terraform_data.release.output}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
terraform_data.release: Refreshing state... [id=026f6bc9-429b-0ee8-47ad-aaec4ef16506]
local_file.release: Refreshing state... [id=3a9c9636f05189eb6c6894a642bc48285378dbf2]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.release must be replaced
-/+ resource "local_file" "release" {
      ~ content              = <<-EOT
            Έκδοση 1.0.0
        EOT -> (known after apply) # forces replacement
      …
      ~ id                   = "3a9c9636f05189eb6c6894a642bc48285378dbf2" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

  # terraform_data.release will be updated in-place
  ~ resource "terraform_data" "release" {
        id               = "026f6bc9-429b-0ee8-47ad-aaec4ef16506"
      ~ input            = "1.0.0" -> "1.1.0"
      ~ output           = "1.0.0" -> (known after apply)
        # (1 unchanged attribute hidden)
    }

Plan: 1 to add, 1 to change, 1 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `terraform_data.release` αλλάζει επί τόπου με `~`. Κρατά το ίδιο `id` και το `output` του θα πάρει νέα τιμή. Το `local_file.release` αντικαθίσταται, γιατί το `content` του εξαρτάται από το `output`. Κάνε apply και άλλαξε μετά μόνο το `triggers_replace`.

TERMINAL

```
terraform apply -auto-approve
```

main.tf

```
resource "terraform_data" "release" {
  input            = "1.1.0"
  triggers_replace = "deploy-2"
}

resource "local_file" "release" {
  filename = "release.txt"
  content  = "Έκδοση ${terraform_data.release.output}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
terraform_data.release: Refreshing state... [id=026f6bc9-429b-0ee8-47ad-aaec4ef16506]
local_file.release: Refreshing state... [id=76845cbe05ceca51c8ae042e13776a43a3be3788]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.release must be replaced
-/+ resource "local_file" "release" {
      ~ content              = <<-EOT
            Έκδοση 1.1.0
        EOT -> (known after apply) # forces replacement
      …
      ~ id                   = "76845cbe05ceca51c8ae042e13776a43a3be3788" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

  # terraform_data.release must be replaced
-/+ resource "terraform_data" "release" {
      ~ id               = "026f6bc9-429b-0ee8-47ad-aaec4ef16506" -> (known after apply)
      ~ output           = "1.1.0" -> (known after apply)
      ~ triggers_replace = "deploy-1" -> "deploy-2"
        # (1 unchanged attribute hidden)
    }

Plan: 2 to add, 0 to change, 2 to destroy.

──────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Τώρα το `terraform_data.release` αντικαθίσταται με `-/+` και παίρνει νέο `id`. Το `local_file.release` αντικαθίσταται κι αυτό, αν και το κείμενο θα μείνει `Έκδοση 1.1.0`. Το `output` του νέου resource γίνεται γνωστό μόνο μετά το apply, οπότε το Terraform δεν μπορεί να εγγυηθεί από πριν ότι θα είναι ίδιο. Η αντικατάσταση ταξιδεύει κατά μήκος των βελών του graph.

Κύκλοι εξαρτήσεων

Αν το A χρειάζεται το B και το B χρειάζεται το A, καμία σειρά δεν ικανοποιεί και τα δύο. Αυτό λέγεται **κύκλος** (cycle). Το Terraform τον εντοπίζει όταν χτίζει το graph, πριν αγγίξει οτιδήποτε.

TERMINAL

```
mkdir ../cycle
cd ../cycle
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "ping" {
  filename = "ping.txt"
  content  = "Η απάντηση είναι στο ${local_file.pong.filename}\n"
}

resource "local_file" "pong" {
  filename = "pong.txt"
  content  = "Η ερώτηση είναι στο ${local_file.ping.filename}\n"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Cycle:
│   local_file.ping
│   local_file.pong
│
╵
```

Το σφάλμα `Cycle` απαριθμεί τα resources που σχηματίζουν τον κύκλο. Το ίδιο σφάλμα δίνουν και τα `plan` και `apply`. Ο κύκλος δεν διορθώνεται με `depends_on`, γιατί το `depends_on` προσθέτει βέλη και δεν αφαιρεί. Τον σπας αφαιρώντας μία από τις αναφορές ή μεταφέροντας την κοινή τιμή σε τρίτο resource που διαβάζουν και τα δύο. Το δεύτερο το κάνεις στην άσκηση 6.6.

Ασκήσεις

Λύσε κάθε άσκηση σε νέο φάκελο μέσα στο `~/tfbook/ch06`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από αντίγραφο υπάρχοντος. Τρέξε `terraform init` πριν από τις άλλες εντολές και δώσε `-auto-approve` στα apply.

ΑΣΚΗΣΗ 6.1 Διευθύνσεις στο state

Γράψε configuration με τρία resources. Ένα `random_pet` με όνομα `db` και τρεις λέξεις. Ένα `random_integer` με όνομα `replicas` από 1 έως 5. Ένα `random_string` με όνομα `suffix`, τέσσερις χαρακτήρες, χωρίς σύμβολα και χωρίς κεφαλαία. Κάνε apply και τύπωσε τις διευθύνσεις των resources.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
random_integer.replicas
random_pet.db
random_string.suffix
```

ΑΣΚΗΣΗ 6.2 Η τυχαία τιμή μένει

Ξεκίνα από αντίγραφο του φακέλου της άσκησης 6.1. Κάνε apply δύο φορές στη σειρά και δες αν άλλαξαν οι τυχαίες τιμές. Μετά άλλαξε το `length` του `random_pet.db` σε 2 και τρέξε `terraform plan`. Τι θα γίνει στο όνομα;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # random_pet.db must be replaced
…
      ~ length    = 3 -> 2 # forces replacement
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 6.3 Η σειρά των blocks δεν μετράει

Γράψε το παρακάτω `main.tf`, όπου τα blocks είναι γραμμένα από το τελευταίο προς το πρώτο. Πριν από το apply γράψε σε χαρτί με ποια σειρά πιστεύεις ότι θα φτιαχτούν τα τρία resources. Μετά κάνε apply.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "local_file" "receipt" {
  filename = "receipt.txt"
  content  = "Ο κωδικός γράφτηκε στο ${local_file.code.filename}\n"
}

resource "local_file" "code" {
  filename = "code.txt"
  content  = "${random_string.code.result}\n"
}

resource "random_string" "code" {
  length = 6
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
random_string.code: Creating...
…
local_file.code: Creating...
…
local_file.receipt: Creating...
…
```

ΑΣΚΗΣΗ 6.4 Από depends_on σε αναφορά

Ξεκίνα από αντίγραφο του φακέλου `backup`. Βγάλε το `depends_on` από το `local_file.backup` και κάνε το `source` αναφορά στο attribute `filename` του `local_file.notes`. Τύπωσε το graph και έλεγξε ότι η εξάρτηση υπάρχει ακόμη. Μετά κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  "local_file.backup" -> "local_file.notes";
  "local_file.notes" -> "random_pet.server";
…
```

ΑΣΚΗΣΗ 6.5 Λάθος στόχος στο depends_on

Ξεκίνα από αντίγραφο του φακέλου `backup`. Άλλαξε το `depends_on` ώστε να δείχνει στο attribute `filename` του `local_file.notes` αντί για ολόκληρο το resource. Τρέξε `terraform validate`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Error: Invalid depends_on reference

  on main.tf line 12, in resource "local_file" "backup":
  12:   depends_on = [local_file.notes.filename]

References in depends_on must be to a whole object (resource, etc), not to
an attribute of an object.
```

ΑΣΚΗΣΗ 6.6 Σπάσε τον κύκλο

Ξεκίνα από αντίγραφο του φακέλου `cycle`. Τα δύο αρχεία πρέπει να γράφουν το ίδιο τυχαίο όνομα παιχνιδιού. Πρόσθεσε ένα `random_pet` με όνομα `game` και κάνε και τα δύο `local_file` να το διαβάζουν, χωρίς να αναφέρει το ένα το άλλο. Τρέξε `terraform validate` και `terraform graph`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  "local_file.ping" -> "random_pet.game";
  "local_file.pong" -> "random_pet.game";
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 6.1

Μια configuration περιέχει το block `resource "random_pet" "db" {}`. Ποια έκφραση δίνει το όνομα που παρήγαγε σε ένα argument άλλου resource;

a. `random_pet.db.id`
b. `random_pet["db"].id`
c. `db.random_pet.id`
d. `random_pet.db.name`

ΕΡΩΤΗΣΗ 6.2

Το `local_file.report` χρησιμοποιεί στο `content` του την τιμή `random_integer.port.result`. Δεν υπάρχει `depends_on`. Τι ισχύει για τη σειρά δημιουργίας;

a. Τα δύο φτιάχνονται παράλληλα, γιατί δεν υπάρχει `depends_on`.
b. Φτιάχνεται πρώτο όποιο είναι γραμμένο πρώτο στο αρχείο.
c. Το `random_integer.port` φτιάχνεται πρώτο, γιατί η αναφορά δημιουργεί implicit dependency.
d. Το apply αποτυγχάνει, γιατί η τιμή είναι `(known after apply)`.

ΕΡΩΤΗΣΗ 6.3

Ένα πρόγραμμα σε server του AWS διαβάζει στην εκκίνηση ένα αντικείμενο από S3 bucket. Κανένα argument του `aws_instance` δεν αναφέρει το bucket ή το αντικείμενο. Πώς εξασφαλίζεις ότι το αντικείμενο θα υπάρχει πριν από τον server;

a. Γράφεις το block του αντικειμένου πριν από το block του server.
b. Προσθέτεις `depends_on` στο `aws_instance` με το resource του αντικειμένου.
c. Τρέχεις το apply με `-parallelism=1`.
d. Δεν χρειάζεται τίποτα, το Terraform φτιάχνει πάντα πρώτα την αποθήκευση.

ΕΡΩΤΗΣΗ 6.4

Το `local_file.summary` αναφέρει το `local_file.config` και αυτό αναφέρει το `random_pet.server`. Με ποια σειρά τα σβήνει το `terraform destroy`;

a. `random_pet.server`, `local_file.config`, `local_file.summary`
b. Και τα τρία παράλληλα, γιατί το destroy δεν κοιτά εξαρτήσεις.
c. Με τη σειρά που είναι γραμμένα στη configuration.
d. `local_file.summary`, `local_file.config`, `random_pet.server`

ΕΡΩΤΗΣΗ 6.5

Πόσες λειτουργίες σε resources εκτελεί το Terraform ταυτόχρονα, αν δεν δώσεις `-parallelism`;

a. 1
b. 10
c. Όσους πυρήνες έχει ο επεξεργαστής
d. Χωρίς όριο

ΕΡΩΤΗΣΗ 6.6

Ποια ισχύουν για το `terraform_data`; Επίλεξε δύο.

a. Είναι ενσωματωμένο στο Terraform και δεν χρειάζεται εγγραφή στο `required_providers`.
b. Μια αλλαγή στο `input` το αντικαθιστά πάντα.
c. Μια αλλαγή στο `triggers_replace` το αντικαθιστά.
d. Φτιάχνει αρχείο στον δίσκο με την τιμή του `input`.



ΜΕΡΟΣ ΙΙ

Η γλώσσα

Στο πρώτο μέρος κάθε τιμή ήταν γραμμένη μέσα στον κώδικα. Σε αυτό το μέρος μαθαίνεις τη γλώσσα που κάνει μια configuration ευέλικτη. Θα δεις μεταβλητές και outputs, τύπους, εκφράσεις και συναρτήσεις, επαναλήψεις, data sources, ελέγχους και τον κύκλο ζωής των resources. Τα resources παραμένουν αρχεία στον δίσκο σου, οπότε η προσοχή σου πηγαίνει στη γλώσσα και όχι στο cloud.



ΚΕΦΑΛΑΙΟ 7

Input variables

Μέχρι τώρα το όνομα και το περιεχόμενο κάθε αρχείου ήταν γραμμένα μέσα στο `main.tf`. Σε αυτό το κεφάλαιο δηλώνεις μεταβλητές, τους δίνεις τιμές από τη γραμμή εντολών, από αρχεία και από το περιβάλλον και μαθαίνεις ποια τιμή κερδίζει όταν η ίδια μεταβλητή έρχεται από πολλές πηγές.

Γιατί χρειάζονται μεταβλητές

Στο κεφάλαιο 1 το `local_file.hello` είχε το `filename` και το `content` γραμμένα μέσα στο block. Για να γράψεις άλλο αρχείο έπρεπε να αλλάξεις τον κώδικα. Σε πραγματική υποδομή αυτό γίνεται γρήγορα πρόβλημα. Η ίδια configuration στήνει ένα μικρό περιβάλλον για δοκιμές και ένα μεγάλο για production. Ανάμεσά τους αλλάζει ένα όνομα, ένα μέγεθος ή μια περιοχή. Αν κρατάς δύο αντίγραφα του κώδικα, κάποια μέρα θα διορθώσεις το ένα και θα ξεχάσεις το άλλο.

Η λύση είναι η **input variable**, στο εξής απλώς μεταβλητή. Είναι μια παράμετρος της configuration, όπως η παράμετρος μιας συνάρτησης σε μια γλώσσα προγραμματισμού. Ο κώδικας λέει «εδώ μπαίνει ένα όνομα αρχείου». Όποιος τρέχει το Terraform αποφασίζει ποιο.

Το block variable

Φτιάξε φάκελο για το παράδειγμα του κεφαλαίου.

TERMINAL

```
mkdir greet
cd greet
```

Γράψε τις μεταβλητές στο `variables.tf`.

variables.tf

```
variable "filename" {
  type        = string
  description = "Το αρχείο που θα γράψει το Terraform"
}

variable "message" {
  type        = string
  description = "Το κείμενο μέσα στο αρχείο"
  default     = "Καλημέρα"
}

variable "pet_length" {
  type        = number
  description = "Πόσες λέξεις έχει το όνομα του κατοικίδιου"
  default     = 2
}
```

Κάθε block `variable` έχει ένα label, το όνομα της μεταβλητής. Το όνομα είναι μοναδικό μέσα στη configuration. Η σύμβαση είναι πεζά γράμματα με κάτω παύλες, όπως στο `pet_length`. Μερικά ονόματα απαγορεύονται γιατί έχουν ειδικό νόημα στο Terraform. Ανάμεσά τους είναι τα `count`, `for_each`, `source`, `version`, `providers`, `depends_on`, `lifecycle` και `locals`. Με `variable "count"` το `terraform validate` απαντά `Invalid variable name`.

Όλα τα arguments μέσα στο block είναι προαιρετικά.

- Το `type` ορίζει τι τιμή δέχεται η μεταβλητή. Εδώ βλέπεις `string` για κείμενο και `number` για αριθμό. Υπάρχει και ο `bool` για τις τιμές `true` και `false`. Οι σύνθετοι τύποι, όπως οι λίστες, είναι το θέμα του κεφαλαίου 9. Χωρίς `type` η μεταβλητή δέχεται οτιδήποτε και μια λάθος τιμή φαίνεται μόνο όταν φτάσει στο resource που τη χρησιμοποιεί. Δήλωνε πάντα τύπο.
- Το `default` είναι η τιμή που παίρνει η μεταβλητή όταν δεν της δώσεις άλλη. Μια μεταβλητή χωρίς `default` είναι **υποχρεωτική**. Το `default` είναι σταθερή τιμή. Αν γράψεις `default = var.message` στο block της `filename`, το Terraform απαντά `Variables not allowed`. Το κεφάλαιο 8 δείχνει το σφάλμα στην πράξη.
- Το `description` εξηγεί σε άνθρωπο τι είναι η μεταβλητή και εμφανίζεται όταν το Terraform ρωτά για τιμή. Γράφε το πάντα.
- Το `nullable` λέει αν η μεταβλητή δέχεται την ειδική τιμή `null`. Το βλέπεις αναλυτικά πιο κάτω.
- Το `sensitive = true` κρύβει την τιμή από την έξοδο του plan και του apply. Δεν την κρύβει από το state. Το κεφάλαιο 24 το εξηγεί μαζί με το `ephemeral`, που κρατά μια τιμή έξω από το state.
- Το εσωτερικό block `validation` ελέγχει την τιμή με έναν κανόνα, για παράδειγμα ότι ένας αριθμός είναι θετικός. Είναι το θέμα του κεφαλαίου 13.

Το όνομα του αρχείου δεν παίζει ρόλο. Όπως είδες στο κεφάλαιο 2, το Terraform διαβάζει όλα τα `.tf` του φακέλου μαζί. Η σύμβαση όμως θέλει τις μεταβλητές στο `variables.tf`, ώστε όποιος ανοίγει το project να βλέπει με μια ματιά τι εισόδους δέχεται.

Χρήση με var

Γράψε τα resources στο `main.tf`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "name" {
  length = var.pet_length
}

resource "local_file" "greeting" {
  filename = var.filename
  content  = var.message
}
```

Μια μεταβλητή διαβάζεται με την έκφραση `var.<όνομα>`. Το `var.filename` δίνει την τιμή της `filename` και το `var.pet_length` την τιμή της `pet_length`. Είναι reference, όπως οι references σε attributes που είδες στο κεφάλαιο 6, με πρόθεμα `var` αντί για τύπο resource. Το `local_file.greeting` παίρνει και το όνομα και το περιεχόμενο από μεταβλητές. Το `random_pet.name` φτιάχνει ένα τυχαίο όνομα κατοικίδιου με τόσες λέξεις όσες λέει η `pet_length`.

Η τιμή μιας μεταβλητής ορίζεται μία φορά πριν ξεκινήσει το plan. Μένει ίδια σε όλη την εκτέλεση. Δεν υπάρχει ανάθεση σε μεταβλητή μέσα στον κώδικα. Για τιμές που υπολογίζονται από άλλες υπάρχουν τα locals του κεφαλαίου 8.

Η configuration χρειάζεται δύο providers. Τρέξε `terraform init`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)
…
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

Το `validate` πέτυχε χωρίς τιμή για το `filename`. Ελέγχει τη σύνταξη και τους τύπους της configuration, όχι τις τιμές που θα δοθούν αργότερα.

Όταν λείπει μια τιμή

Το `filename` δεν έχει `default`. Αν τρέξεις plan χωρίς να του δώσεις τιμή, το Terraform σε ρωτά.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
var.filename
  Το αρχείο που θα γράψει το Terraform

  Enter a value: note.txt


Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.greeting will be created
  + resource "local_file" "greeting" {
      + content              = "Καλημέρα"
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "note.txt"
      + id                   = (known after apply)
    }

  # random_pet.name will be created
  + resource "random_pet" "name" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 2 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η ερώτηση γράφει το όνομα `var.filename` και από κάτω το description. Απάντησες `note.txt`. Στο plan το `filename` έχει αυτή την τιμή. Το `content` είναι `"Καλημέρα"`, το `default` της `message`, γιατί δεν δόθηκε άλλη τιμή.

Το `length` είναι `2` για τον ίδιο λόγο. Για τη `message` και την `pet_length` το Terraform δεν ρώτησε. Ρωτά μόνο για υποχρεωτικές μεταβλητές που δεν πήραν τιμή από πουθενά.

Η ερώτηση βολεύει όταν δοκιμάζεις κάτι στο terminal. Σε ένα pipeline CI δεν υπάρχει άνθρωπος να απαντήσει. Εκεί τρέχεις το Terraform με `-input=false`, που απαγορεύει κάθε ερώτηση.

TERMINAL

```
terraform plan -input=false
```

ΕΞΟΔΟΣ

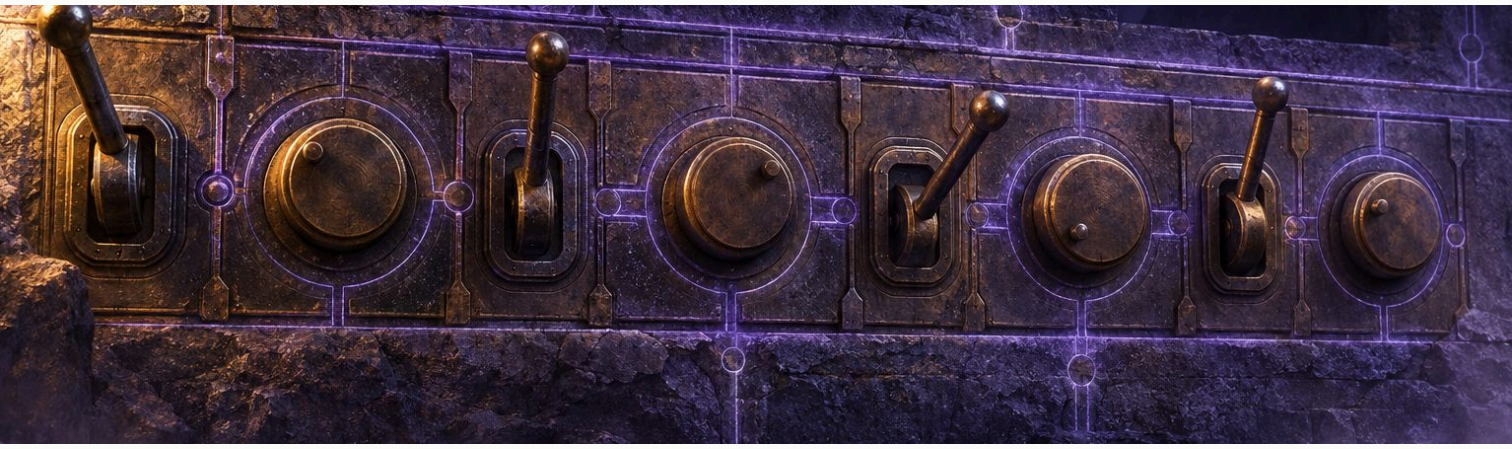
```
╷
│ Error: No value for required variable
│
│   on variables.tf line 1:
│    1: variable "filename" {
│
│ The root module input variable "filename" is not set, and has no default
│ value. Use a -var or -var-file command line argument to provide a value for
│ this variable.
```

Τώρα η υποχρεωτική μεταβλητή μένει χωρίς τιμή και το plan σταματά με `No value for required variable`. Το μήνυμα δείχνει το αρχείο και τη γραμμή της δήλωσης και προτείνει τη λύση, ένα `-var` ή ένα `-var-file`.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Μεταβλητή χωρίς `default` είναι υποχρεωτική. Αν δεν πάρει τιμή από καμία πηγή, το Terraform ρωτά. Με `-input=false` δεν ρωτά και σταματά με σφάλμα.

Τιμές από τη γραμμή εντολών

Το `-var` δίνει τιμή σε μία μεταβλητή με τη μορφή `όνομα=τιμή`. Το επαναλαμβάνεις για όσες μεταβλητές θέλεις.



TERMINAL

```
terraform apply -auto-approve -var filename=note.txt -var 'message=Καλημέρα κόσμε'
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.greeting will be created
  + resource "local_file" "greeting" {
      + content              = "Καλημέρα κόσμε"
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "note.txt"
      + id                   = (known after apply)
    }

  # random_pet.name will be created
  + resource "random_pet" "name" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 2 to add, 0 to change, 0 to destroy.
local_file.greeting: Creating...
random_pet.name: Creating...
random_pet.name: Creation complete after 0s [id=sincere-iguana]
local_file.greeting: Creation complete after 0s [id=71b3f788c54f100a5206c35c104ea773b8adcbe5]

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

TERMINAL

```
cat note.txt
```

ΕΞΟΔΟΣ

```
Καλημέρα κόσμε
```

Τα μονά εισαγωγικά γύρω από το δεύτερο `-var` είναι για το shell. Χωρίς αυτά το shell θα έκοβε την τιμή στο κενό. Το Terraform δεν τα βλέπει. Το `random_pet.name` πήρε δύο λέξεις, όσες λέει το `default`. Το δικό σου όνομα θα είναι διαφορετικό, γιατί είναι τυχαίο. Το `note.txt` δεν έχει αλλαγή γραμμής στο τέλος, γιατί το `content` είναι ακριβώς η τιμή της μεταβλητής.

Οι τιμές του `-var` ισχύουν μόνο για την εντολή όπου τις γράφεις. Δες τι γίνεται αν στο επόμενο plan ξεχάσεις τη `message`.

TERMINAL

```
terraform plan -var filename=note.txt
```

ΕΞΟΔΟΣ

```
…
  # local_file.greeting must be replaced
…
      ~ content              = "Καλημέρα κόσμε" -> "Καλημέρα" # forces replacement
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Η `message` γύρισε στο `default` της. Το Terraform δεν θυμάται ότι την προηγούμενη φορά ήταν `Καλημέρα κόσμε`. Βλέπει ότι η configuration ζητά `Καλημέρα` και σχεδιάζει να αντικαταστήσει το αρχείο.

ΠΑΓΙΔΑ

Κάθε `plan`, `apply` και `destroy` χρειάζεται ξανά τις τιμές των μεταβλητών, ακόμη και το `destroy` που απλώς σβήνει. Αν ξεχάσεις ένα `-var`, το Terraform είτε ρωτά είτε χρησιμοποιεί το `default` και σχεδιάζει αλλαγές που δεν ήθελες. Τιμές που επαναλαμβάνεις σε κάθε εντολή ανήκουν σε αρχείο.

Μεταβλητές περιβάλλοντος

Μια μεταβλητή περιβάλλοντος με όνομα `TF_VAR_` και μετά το όνομα της μεταβλητής δίνει κι αυτή τιμή. Το `TF_VAR_message` δίνει τιμή στη `message`. Σε Linux και macOS το όνομα μετά το `TF_VAR_` ταιριάζει ακριβώς, με τα ίδια πεζά και κεφαλαία. Το `TF_VAR_MESSAGE` δεν δίνει τιμή σε τίποτα.

TERMINAL

```
export TF_VAR_filename=note.txt
export TF_VAR_message='Καλημέρα κόσμε'
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.name: Refreshing state... [id=sincere-iguana]
local_file.greeting: Refreshing state... [id=71b3f788c54f100a5206c35c104ea773b8adcbe5]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Δεν υπάρχει ερώτηση και δεν υπάρχει αλλαγή. Οι δύο μεταβλητές πήραν τις ίδιες τιμές με το apply, οπότε η configuration συμφωνεί με το state. Το Terraform αγνοεί σιωπηλά ένα `TF_VAR_` για μεταβλητή που δεν δηλώθηκε, ενώ ένα `-var` για αδήλωτη μεταβλητή είναι σφάλμα.

Οι μεταβλητές περιβάλλοντος βολεύουν σε pipelines, όπου το σύστημα CI τις ορίζει πριν τρέξει το Terraform. Έχουν όμως ένα μειονέκτημα. Δεν φαίνονται στην εντολή, οπότε ξεχνάς ότι υπάρχουν. Σβήσε τες, για να μην επηρεάζουν τα επόμενα παραδείγματα.

TERMINAL

```
unset TF_VAR_filename TF_VAR_message
```

Αρχεία μεταβλητών

Όταν οι τιμές είναι πολλές ή τις χρειάζεσαι σε κάθε εντολή, τις γράφεις σε ένα **αρχείο μεταβλητών**. Έχει κατάληξη `.tfvars` και περιέχει μόνο αναθέσεις της μορφής `όνομα = τιμή`.

dev.tfvars

```
filename   = "dev.txt"
message    = "Καλημέρα από το dev"
pet_length = 3
```

Το δίνεις στο Terraform με `-var-file`.

TERMINAL

```
terraform plan -var-file=dev.tfvars
```

ΕΞΟΔΟΣ

```
…
  # local_file.greeting must be replaced
…
      ~ content              = "Καλημέρα κόσμε" -> "Καλημέρα από το dev" # forces replacement
…
      ~ filename             = "note.txt" -> "dev.txt" # forces replacement
…
  # random_pet.name must be replaced
…
      ~ length    = 2 -> 3 # forces replacement
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Και οι τρεις τιμές ήρθαν από το αρχείο. Το plan αντικαθιστά και τα δύο resources, γιατί άλλαξαν το όνομα, το περιεχόμενο και το μήκος. Με ένα `prod.tfvars` δίπλα στο `dev.tfvars` ο ίδιος κώδικας στήνει δύο διαφορετικά περιβάλλοντα.

Ένα αρχείο `.tfvars` γράφεται σε HCL, αλλά δέχεται μόνο σταθερές τιμές. Αν γράψεις μέσα του `message = var.filename`, το Terraform απαντά `Variables not allowed`. Το αρχείο μεταβλητών δεν είναι configuration. Κρατά μόνο δεδομένα. Αν δίνει τιμή σε μεταβλητή που δεν δηλώθηκε, το Terraform συνεχίζει με την προειδοποίηση `Value for undeclared variable`.

Αρχεία που φορτώνονται μόνα τους

Μερικά αρχεία το Terraform τα διαβάζει χωρίς `-var-file`, αρκεί να βρίσκονται στον φάκελο εργασίας.

- Το `terraform.tfvars`.
- Το `terraform.tfvars.json`.
- Κάθε αρχείο που τελειώνει σε `.auto.tfvars` ή `.auto.tfvars.json`.

Κάθε άλλο όνομα, όπως το `dev.tfvars`, διαβάζεται μόνο με `-var-file`. Γράψε ένα `terraform.tfvars` με τις τιμές του apply.

terraform.tfvars

```
filename = "note.txt"
message  = "Καλημέρα κόσμε"
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.name: Refreshing state... [id=sincere-iguana]
local_file.greeting: Refreshing state... [id=71b3f788c54f100a5206c35c104ea773b8adcbe5]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Χωρίς καμία σημαία το Terraform βρήκε τις τιμές στο `terraform.tfvars` και δεν έχει αλλαγές να κάνει. Η `pet_length` πήρε το `default` της.

Το `terraform.tfvars.json` κάνει την ίδια δουλειά σε JSON. Τα κλειδιά του αντικειμένου είναι τα ονόματα των μεταβλητών. Βολεύει όταν τις τιμές τις γράφει άλλο πρόγραμμα, γιατί σχεδόν κάθε γλώσσα παράγει JSON.

terraform.tfvars.json

```
{
  "message": "Καλησπέρα από το JSON"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      ~ content              = "Καλημέρα κόσμε" -> "Καλησπέρα από το JSON" # forces replacement
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Τώρα η `message` έχει τιμή και στα δύο αρχεία και κέρδισε το JSON. Το `filename` συνεχίζει να έρχεται από το `terraform.tfvars`, γιατί το JSON δεν το ορίζει. Κάθε μεταβλητή παίρνει την τιμή της χωριστά, από την πιο ισχυρή πηγή που την ορίζει.

Τα αρχεία `*.auto.tfvars` βολεύουν όταν θέλεις να μοιράσεις τις τιμές σε θεματικά αρχεία, για παράδειγμα ένα για το δίκτυο και ένα για τις βάσεις.

pets.auto.tfvars

```
pet_length = 4
```



TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.greeting must be replaced
…
  # random_pet.name must be replaced
…
      ~ length    = 2 -> 4 # forces replacement
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Το `pets.auto.tfvars` φορτώθηκε μόνο του και η `pet_length` έγινε `4`. Το plan αντικαθιστά πλέον και τα δύο resources.

Η σειρά προτεραιότητας

Είδες ότι μια μεταβλητή μπορεί να πάρει τιμή από πολλά μέρη. Όταν η ίδια μεταβλητή έρχεται από πολλά, το Terraform τα διαβάζει με σταθερή σειρά και κάθε νέα τιμή γράφει πάνω στην προηγούμενη. Κερδίζει η τελευταία. Θα το αποδείξεις σε καθαρό φάκελο, με μία μεταβλητή που παίρνει ως τιμή το όνομα της πηγής της.

TERMINAL

```
cd ..
mkdir layers
cd layers
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "color" {
  type    = string
  default = "default"
}

resource "local_file" "color" {
  filename = "color.txt"
  content  = var.color
}
```

TERMINAL

```
terraform init
```

Χωρίς καμία πηγή ισχύει το `default`.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      + content              = "default"
…
```

Πρόσθεσε μια μεταβλητή περιβάλλοντος.

TERMINAL

```
export TF_VAR_color=env
terraform plan
```

ΕΞΟΔΟΣ

```
…
      + content              = "env"
…
```

Η μεταβλητή περιβάλλοντος πήρε τη θέση του `default`. Τώρα γράψε το `terraform.tfvars`, ενώ η μεταβλητή περιβάλλοντος μένει.

terraform.tfvars

```
color = "terraform.tfvars"
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      + content              = "terraform.tfvars"
…
```

Το αρχείο νίκησε τη μεταβλητή περιβάλλοντος. Πρόσθεσε το JSON.

terraform.tfvars.json

```
{
  "color": "terraform.tfvars.json"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      + content              = "terraform.tfvars.json"
…
```

Πρόσθεσε δύο αρχεία `.auto.tfvars`, ένα με όνομα `a` και ένα με όνομα `b`.

b.auto.tfvars

```
color = "b.auto.tfvars"
```

a.auto.tfvars

```
color = "a.auto.tfvars"
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      + content              = "b.auto.tfvars"
…
```

Τα δύο `.auto.tfvars` νίκησαν το JSON. Ανάμεσά τους κέρδισε το `b`, αν και το έγραψες πρώτο. Τα αρχεία `.auto.tfvars` διαβάζονται με αλφαβητική σειρά ονόματος. Το `a` διαβάστηκε πρώτο και το `b` έγραψε από πάνω του. Τα `.auto.tfvars.json` μπαίνουν στην ίδια αλφαβητική σειρά.

Μένει η γραμμή εντολών. Φτιάξε ένα αρχείο για `-var-file` και δώσε το μαζί με ένα `-var`, με δύο διαφορετικές σειρές.

extra.tfvars

```
color = "extra.tfvars"
```

TERMINAL

```
terraform plan -var color=cli -var-file=extra.tfvars
```

ΕΞΟΔΟΣ

```
…
      + content              = "extra.tfvars"
…
```

TERMINAL

```
terraform plan -var-file=extra.tfvars -var color=cli
```

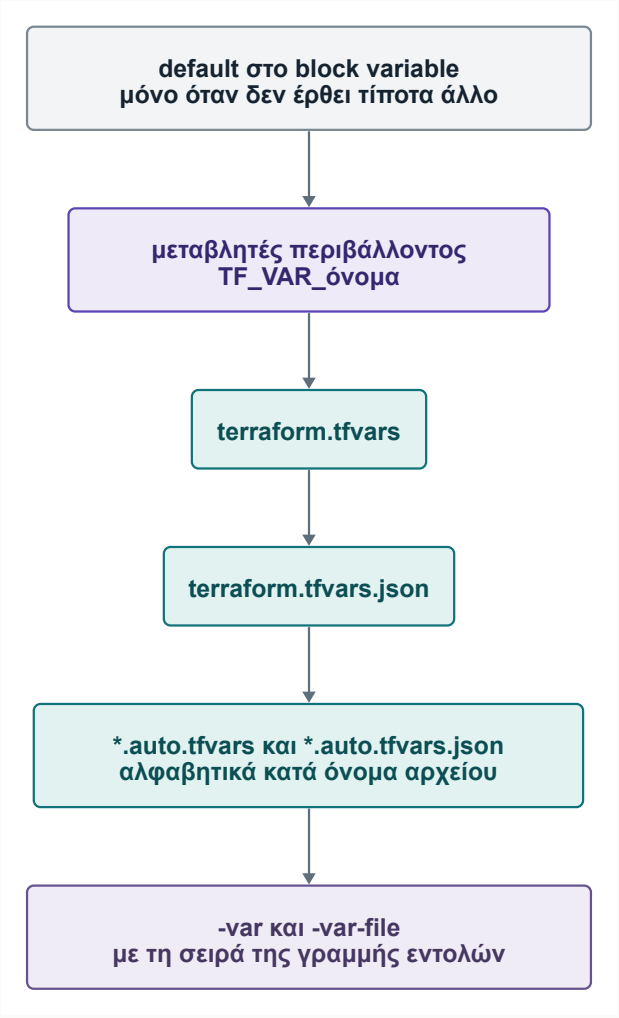
ΕΞΟΔΟΣ

```
…
      + content              = "cli"
…
```

Το `-var` και το `-var-file` νικούν όλες τις άλλες πηγές. Μεταξύ τους δεν έχουν σταθερή ιεραρχία. Διαβάζονται με τη σειρά που τα γράφεις και κερδίζει το τελευταίο. Σβήσε τη μεταβλητή περιβάλλοντος.

TERMINAL

```
unset TF_VAR_color
```



Σχήμα 7.1 · Η σειρά με την οποία το Terraform διαβάζει τιμές για μια μεταβλητή. Κάθε πηγή πιο κάτω γράφει πάνω από όσες βρίσκονται πιο πάνω, οπότε κερδίζει η τελευταία που ορίζει τη μεταβλητή. Το default ισχύει μόνο όταν δεν έρθει τιμή από πουθενά.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά συχνά τη σειρά προτεραιότητας. Από τη χαμηλότερη στην υψηλότερη είναι οι μεταβλητές περιβάλλοντος `TF_VAR_`, το `terraform.tfvars`, το `terraform.tfvars.json`, τα `*.auto.tfvars` και `*.auto.tfvars.json` αλφαβητικά και στο τέλος τα `-var` και `-var-file` με τη σειρά της γραμμής εντολών. Το `default` δεν είναι πηγή που ανταγωνίζεται τις άλλες. Ισχύει μόνο όταν όλες λείπουν. Η ερώτηση για τιμή γίνεται μόνο όταν λείπει και το `default`.

Τύποι και αυτόματες μετατροπές

Γύρνα στον φάκελο `greet`. Η `pet_length` είναι τύπου `number`. Δώσε της τιμή που δεν είναι αριθμός.

TERMINAL

```
cd ../greet
```

TERMINAL

```
terraform plan -var pet_length=τρία
```

ΕΞΟΔΟΣ

```
…
Terraform planned the following actions, but then encountered a problem:
…
╷
│ Error: Invalid value for input variable
│
│   on variables.tf line 12:
│   12: variable "pet_length" {
│
│ Unsuitable value for var.pet_length set using -var="pet_length=...": a
│ number is required.
╵
```

Το σφάλμα είναι `Invalid value for input variable`. Δείχνει τη δήλωση της μεταβλητής και την πηγή της λάθος τιμής, εδώ το `-var`. Λέει επίσης τι περίμενε, `a number is required`. Ο τύπος σε προστάτεψε. Η λάθος τιμή σταμάτησε στην είσοδο και δεν έφτασε ποτέ στο `random_pet.name`.

Πάνω από το σφάλμα η γραμμή `Terraform planned the following actions, but then encountered a problem` λέει ότι το Terraform σχεδίασε πρώτα όσα δεν εξαρτώνται από την `pet_length`. Στη θέση του `…` που την ακολουθεί βρίσκεται η αντικατάσταση του `local_file.greeting`, που ζητά το JSON της προηγούμενης ενότητας. Ένα plan με σφάλμα δεν εκτελείται, οπότε αυτό το μισό plan είναι μόνο πληροφορία.

Δώσε τώρα αριθμό.

TERMINAL

```
terraform plan -var pet_length=3
```

ΕΞΟΔΟΣ

```
…
      ~ length    = 2 -> 3 # forces replacement
…
```

Από τη γραμμή εντολών και από το περιβάλλον κάθε τιμή φτάνει ως κείμενο. Το `3` εδώ ήταν το string `"3"`. Το Terraform το μετέτρεψε σε αριθμό, γιατί αυτό ζητά ο τύπος της μεταβλητής. Το ίδιο κάνει για τον `bool`, όπου τα strings `"true"` και `"false"` γίνονται `true` και `false`. Η μετατροπή γίνεται και μέσα στα αρχεία. Ένα `pet_length = "3"` στο `terraform.tfvars` δίνει τον αριθμό `3`.

ΣΗΜΕΙΩΣΗ

Για σύνθετους τύπους η τιμή στη γραμμή εντολών γράφεται με σύνταξη HCL. Μια μεταβλητή με `type = list(string)` παίρνει τιμή με `-var 'names=["Άννα","Νίκος"]'`. Οι λίστες και οι υπόλοιποι σύνθετοι τύποι εξηγούνται στο κεφάλαιο 9.

Η τιμή null και το nullable

Το `null` σημαίνει «καμία τιμή». Ένα argument με τιμή `null` είναι σαν να μην το έγραψες καθόλου. Το κεφάλαιο 9 το εξετάζει μαζί με τους τύπους. Εδώ μας ενδιαφέρει τι γίνεται όταν δίνεις `null` σε μεταβλητή. Από τη γραμμή εντολών δεν γίνεται, γιατί το `-var message=null` δίνει τη λέξη `"null"`. Σε αρχείο μεταβλητών γίνεται.

null.tfvars

```
message = null
```

TERMINAL

```
terraform plan -var-file=null.tfvars
```

ΕΞΟΔΟΣ

```
…
Terraform planned the following actions, but then encountered a problem:
…
│ Error: Invalid Attribute Combination
│
│   with local_file.greeting,
│   on main.tf line 18, in resource "local_file" "greeting":
│   18: resource "local_file" "greeting" {
│
│ No attribute specified when one (and only one) of
│ [content,content_base64,source] is required
│
…
```

Η `message` έχει `default`, αλλά δεν χρησιμοποιήθηκε. Το `null` είναι κανονική τιμή για μια μεταβλητή, οπότε έφτασε στο `content`. Ο provider `local` θέλει περιεχόμενο και αρνήθηκε με `Invalid Attribute Combination`. Στη θέση του τελευταίου … ακολουθούν άλλα τρία παρόμοια μηνύματα για το ίδιο λάθος. Το μισό plan πάνω από το σφάλμα είναι η αντικατάσταση του `random_pet.name`, γιατί το `pets.auto.tfvars` ζητά τέσσερις λέξεις.

Αυτή είναι η προεπιλογή, `nullable = true`. Με `nullable = false` η μεταβλητή δεν δέχεται ποτέ `null`. Αν έχει `default`, ένα `null` την κάνει να πάρει το `default`. Αν δεν έχει, το `null` είναι σφάλμα. Πρόσθεσε το `nullable = false` στη `message`.

variables.tf

```
variable "filename" {
  type        = string
  description = "Το αρχείο που θα γράψει το Terraform"
}

variable "message" {
  type        = string
  description = "Το κείμενο μέσα στο αρχείο"
  default     = "Καλημέρα"
  nullable    = false
}

variable "pet_length" {
  type        = number
  description = "Πόσες λέξεις έχει το όνομα του κατοικίδιου"
  default     = 2
}
```

TERMINAL

```
terraform plan -var-file=null.tfvars
```

ΕΞΟΔΟΣ

```
…
      ~ content              = "Καλημέρα κόσμε" -> "Καλημέρα" # forces replacement
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Το plan πέτυχε και το `content` θα γίνει `Καλημέρα`, το `default`. Πρόσεξε ότι δεν έγινε `Καλησπέρα από το JSON`. Το `null` ήρθε από την ισχυρότερη πηγή, το `-var-file`, οπότε έγραψε πάνω από το JSON. Μετά το `nullable = false` το αντικατέστησε με το `default`.

> **ΠΑΓΙΔΑ**
> Ένα `null` δεν σημαίνει «αγνόησε αυτή την πηγή και ψάξε στις επόμενες». Είναι τιμή και νικά όπως κάθε άλλη. Με την προεπιλογή `nullable = true` φτάνει μέχρι τα resources, ακόμη κι αν η μεταβλητή έχει `default`.

Μεταβλητές, secrets και Git

Τα αρχεία μεταβλητών είναι βολικά και γι' αυτό καταλήγουν να κρατούν τα πάντα, ακόμη και κωδικούς βάσεων ή API tokens. Ένα τέτοιο αρχείο δεν μπαίνει σε Git. Ό,τι γίνει commit το βλέπει όποιος έχει πρόσβαση στο repository. Μένει επίσης στο ιστορικό, ακόμη κι αν σβήσεις το αρχείο στο επόμενο commit. Αν ένας κωδικός μπει σε Git, τον θεωρείς δημόσιο και τον αλλάζεις.

Η συνηθισμένη πρακτική έχει τρία κομμάτια. Το `.gitignore` κρατά έξω τα αρχεία μεταβλητών. Ένα αρχείο με παράδειγμα, όπως το `terraform.tfvars.example`, μπαίνει σε Git με ψεύτικες τιμές, ώστε ο επόμενος να ξέρει τι πρέπει να συμπληρώσει. Οι πραγματικοί κωδικοί φτάνουν την ώρα της εκτέλεσης, συνήθως ως `TF_VAR_` από τα secrets του συστήματος CI ή από ένα εργαλείο όπως το Vault του κεφαλαίου 24.

Πριν δεις το Git, καθάρισε το `greet`. Το `destroy` χρειάζεται κι αυτό το `filename`, που το βρίσκει στο `terraform.tfvars`.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 2 destroyed.
```

Γράψε ένα `.gitignore` και δες τι θα κρατούσε έξω το Git.

.gitignore

```
.terraform/
*.tfstate
*.tfstate.*
*.tfvars
*.tfvars.json
```

TERMINAL

```
git init -q
git status --short --ignored
```

ΕΞΟΔΟΣ

```
?? .gitignore
?? .terraform.lock.hcl
?? main.tf
?? variables.tf
!! .terraform/
!! dev.tfvars
!! null.tfvars
!! pets.auto.tfvars
!! terraform.tfstate
!! terraform.tfstate.backup
!! terraform.tfvars
!! terraform.tfvars.json
```

Το `git status --short --ignored` δείχνει με `??` τα αρχεία που μπορούν να μπουν σε commit και με `!!` όσα αγνοεί. Μπαίνουν η configuration, το `.gitignore` και το lock file. Μένουν έξω ο φάκελος `.terraform`, τα αρχεία του state και όλα τα αρχεία μεταβλητών. Το `pets.auto.tfvars` πιάνεται από το `*.tfvars`, γιατί κι αυτό τελειώνει σε `.tfvars`. Αν θέλεις να μπει σε Git ένα αρχείο χωρίς secrets, όπως το `dev.tfvars`, πρόσθεσε στο `.gitignore` τη γραμμή `!dev.tfvars`.

Το state μένει έξω για τον ίδιο λόγο. Όπως είδες στο κεφάλαιο 4, κρατά τις τιμές των attributes σε απλό κείμενο. Ένα secret που πέρασε σε resource βρίσκεται εκεί ό,τι κι αν έκανες με το αρχείο μεταβλητών.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch07`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από άλλον. Τρέξε `terraform init` πριν από τις εντολές που ζητά η εκφώνηση.

ΑΣΚΗΣΗ 7.1 Η πόλη σου

Γράψε configuration με τον provider `local` και μία μεταβλητή `city` τύπου `string`, χωρίς `default`, με description. Ένα `local_file` με όνομα `city` γράφει το αρχείο `city.txt` με περιεχόμενο την τιμή της μεταβλητής. Κάνε apply με `-auto-approve` και δώσε την τιμή `Θεσσαλονίκη` από τη γραμμή εντολών. Μετά δείξε το αρχείο με `cat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Θεσσαλονίκη
```

ΑΣΚΗΣΗ 7.2 Δύο περιβάλλοντα

Ο φάκελος της άσκησης ξεκινά ως αντίγραφο του φακέλου της άσκησης 7.1, χωρίς state και χωρίς `.terraform`. Γράψε δύο αρχεία μεταβλητών, το `dev.tfvars` με την πόλη `Λάρισα` και το `prod.tfvars` με την πόλη `Αθήνα`. Τρέξε `terraform plan` μία φορά με το καθένα. Το αναμενόμενο αποτέλεσμα δείχνει τη γραμμή του `content` στα δύο plans.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      + content              = "Λάρισα"
…
      + content              = "Αθήνα"
…
```

ΑΣΚΗΣΗ 7.3 Λέξεις με αριθμό

Γράψε configuration με τον provider `random` και μια μεταβλητή `words` τύπου `number` με `default` το `2`. Ένα `random_pet` με όνομα `pet` παίρνει το `length` από τη `words`. Τρέξε `terraform plan` πρώτα με `-var words=πέντε` και μετά με `-var words=5`. Το αναμενόμενο αποτέλεσμα είναι η γραμμή του `length` στο δεύτερο plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      + length    = 5
…
```

ΑΣΚΗΣΗ 7.4 Ένα null που δεν περνά

Γράψε σε νέο φάκελο το `main.tf` και το `terraform.tfvars` που ακολουθούν. Το `terraform plan` αποτυγχάνει, γιατί το `null` φτάνει στο `content`. Άλλαξε μόνο το block `variable` ώστε το plan να γράφει στο αρχείο το `default`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "message" {
  type    = string
  default = "Γεια σου"
}

resource "local_file" "hello" {
  filename = "hello.txt"
  content  = var.message
}
```

terraform.tfvars

```
message = null
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      + content              = "Γεια σου"
…
```

ΑΣΚΗΣΗ 7.5 Ένα token έξω από το Git

Γράψε configuration με τον provider `local`, μια μεταβλητή `api_token` τύπου `string` χωρίς `default` και ένα `local_file` με όνομα `config` που γράφει το `config.txt` με περιεχόμενο το token. Δώσε την τιμή `abc123` στο αρχείο `secret.auto.tfvars`. Γράψε `.gitignore` που κρατά έξω τον φάκελο `.terraform`, τα αρχεία του state και κάθε αρχείο μεταβλητών. Τρέξε `terraform plan -input=false` για να δεις ότι η τιμή βρέθηκε και μετά `git init -q` και `git status --short --ignored`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
?? .gitignore
?? .terraform.lock.hcl
?? main.tf
!! .terraform/
!! secret.auto.tfvars
```

ΑΣΚΗΣΗ 7.6 Ποιος κερδίζει

Γράψε σε νέο φάκελο τα αρχεία που ακολουθούν. Πριν τρέξεις οτιδήποτε, πρόβλεψε τι θα γράψει το `content` σε δύο plans, ενώ ισχύει η μεταβλητή περιβάλλοντος `TF_VAR_color=μαύρο`. Το πρώτο είναι σκέτο `terraform plan`. Το δεύτερο είναι `terraform plan -var color=λευκό -var-file=extra.tfvars`. Μετά τρέξε τα δύο plans και σβήσε τη μεταβλητή περιβάλλοντος.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "color" {
  type    = string
  default = "γκρι"
}

resource "local_file" "color" {
  filename = "color.txt"
  content  = var.color
}
```

terraform.tfvars

```
color = "πράσινο"
```

z.auto.tfvars

```
color = "κόκκινο"
```

a.auto.tfvars

```
color = "μπλε"
```

extra.tfvars

```
color = "κίτρινο"
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      + content              = "κόκκινο"
…
      + content              = "κίτρινο"
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 7.1

Η μεταβλητή `region` παίρνει τιμή από το `terraform.tfvars`, από το `prod.auto.tfvars`, από τη μεταβλητή περιβάλλοντος `TF_VAR_region` και από το `-var-file=override.tfvars` της εντολής. Ποια τιμή χρησιμοποιεί το Terraform;

a. Του `terraform.tfvars`
b. Του `prod.auto.tfvars`
c. Της `TF_VAR_region`
d. Του `override.tfvars`

ΕΡΩΤΗΣΗ 7.2

Ποια αρχεία με τιμές μεταβλητών φορτώνει το Terraform μόνο του, χωρίς `-var-file`; Επίλεξε δύο.

a. `terraform.tfvars`
b. `prod.tfvars`
c. `network.auto.tfvars`
d. `values.tfvars.json`

ΕΡΩΤΗΣΗ 7.3

Η μεταβλητή `db_name` δεν έχει `default` και καμία πηγή δεν της δίνει τιμή. Το pipeline τρέχει `terraform apply -input=false -auto-approve`. Τι γίνεται;

a. Η μεταβλητή παίρνει κενό string και το apply συνεχίζει.
b. Το Terraform περιμένει να πληκτρολογήσει κάποιος την τιμή.
c. Το Terraform σταματά με σφάλμα ότι η υποχρεωτική μεταβλητή δεν έχει τιμή.
d. Το Terraform παραλείπει τα resources που χρησιμοποιούν τη μεταβλητή.

ΕΡΩΤΗΣΗ 7.4

Θέλεις να δώσεις τιμή στη μεταβλητή `instance_type` από το περιβάλλον σε Linux. Ποια μεταβλητή περιβάλλοντος ορίζεις;

a. `TF_instance_type`
b. `TF_VAR_INSTANCE_TYPE`
c. `TF_VAR_instance_type`
d. `TERRAFORM_VAR_instance_type`

ΕΡΩΤΗΣΗ 7.5

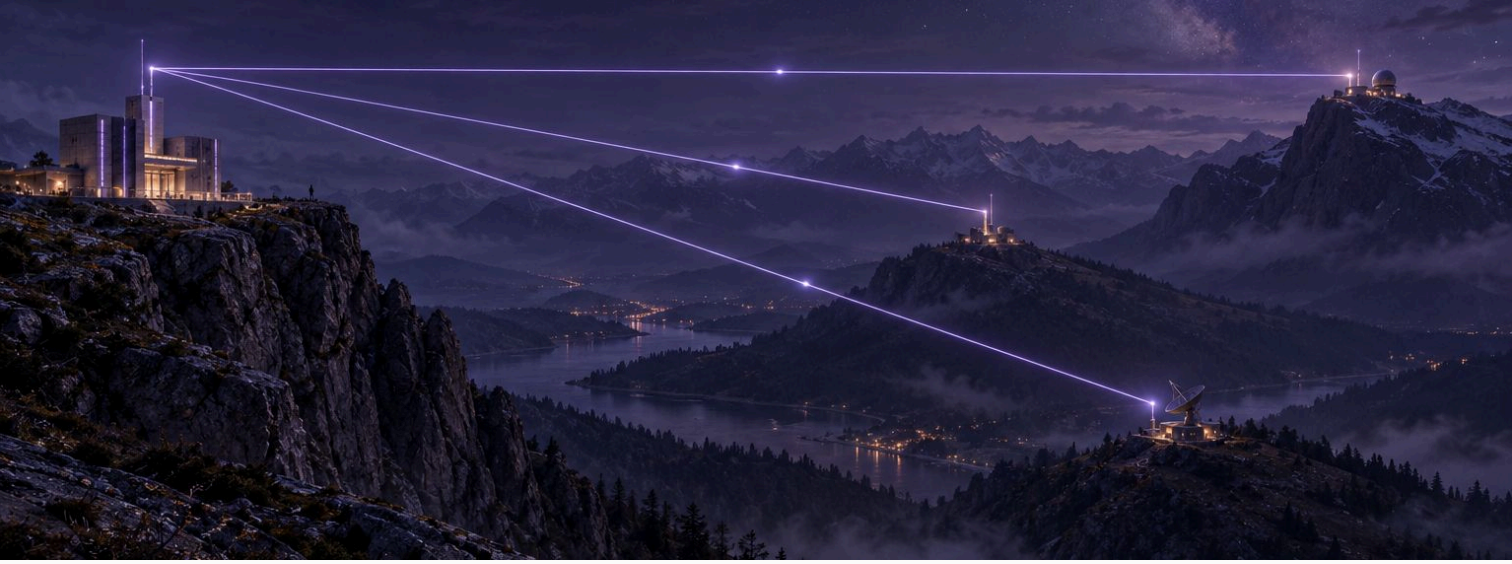
Σωστό ή λάθος; Μια μεταβλητή με `default` και `nullable = false` παίρνει το `default` όταν ένα αρχείο μεταβλητών της δίνει `null`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 7.6

Ένα αρχείο `secrets.tfvars` κρατά τον κωδικό μιας βάσης δεδομένων. Ποια είναι η σωστή πρακτική;

a. Μπαίνει σε Git, ώστε να το έχει όλη η ομάδα.
b. Μένει έξω από το Git και ο κωδικός δίνεται την ώρα της εκτέλεσης, για παράδειγμα με `TF_VAR_`.
c. Μετονομάζεται σε `secrets.auto.tfvars` και μπαίνει σε Git, ώστε να μη χρειάζεται `-var-file`.
d. Η μεταβλητή παίρνει `sensitive = true` και τότε το αρχείο μπαίνει με ασφάλεια σε Git.



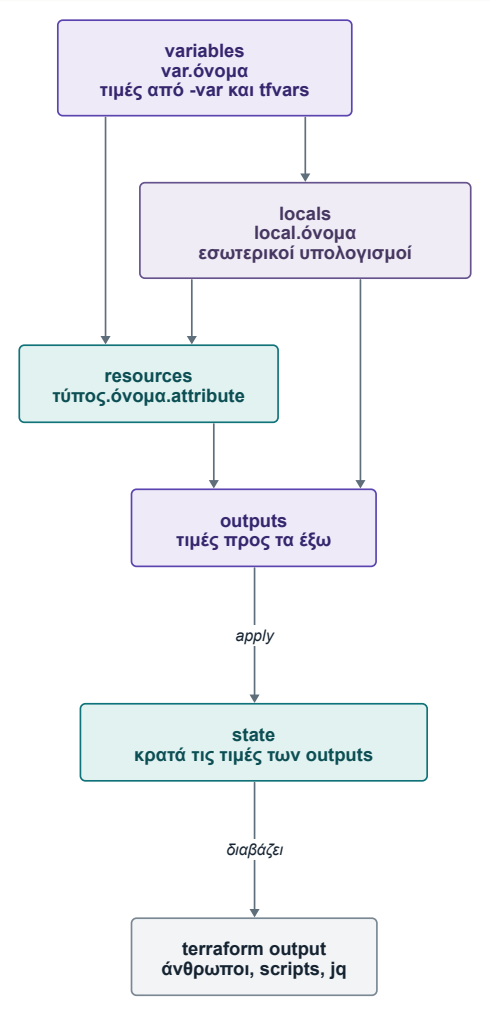
ΚΕΦΑΛΑΙΟ 8

Outputs και locals

Οι μεταβλητές του κεφαλαίου 7 φέρνουν τιμές μέσα στη configuration. Εδώ γνωρίζεις τις άλλες δύο μορφές τιμών. Τα outputs βγάζουν τιμές προς τα έξω, για σένα και για τα scripts σου. Τα locals δίνουν όνομα σε υπολογισμούς που χρειάζεσαι σε πολλά σημεία της configuration.

Είσοδος, υπολογισμός, έξοδος

Μια configuration μοιάζει με μια συνάρτηση σε γλώσσα προγραμματισμού. Οι input variables είναι οι παράμετροί της. Τις δίνει όποιος την τρέχει, με `-var`, με αρχεία `.tfvars` ή αφήνοντας τις προεπιλογές. Τα **locals** είναι οι τοπικές μεταβλητές μέσα στο σώμα της. Υπολογίζονται από τις παραμέτρους και από άλλες τιμές και κανείς δεν τα αλλάζει απ' έξω. Τα **outputs** είναι οι τιμές που επιστρέφει. Μετά το apply τις διαβάζεις εσύ, ένα script ή μια άλλη configuration.



Σχήμα 8.1 · Οι τιμές μιας configuration. Οι variables έρχονται απ' έξω, τα locals υπολογίζονται μέσα και τα outputs βγαίνουν προς τα έξω. Το apply γράφει τα outputs στο state και το terraform output τα διαβάζει από εκεί.

Το σχήμα δείχνει και κάτι που θα δεις στην πράξη σε λίγο. Τα outputs δεν υπολογίζονται κάθε φορά που τα ζητάς. Το apply τα γράφει στο state και η εντολή `terraform output` τα διαβάζει από εκεί.

Το block output

Φτιάξε τον φάκελο του παραδείγματος.

TERMINAL

```
mkdir shop
cd shop
```

Οι δύο μεταβλητές μπαίνουν στο `variables.tf`, όπως έμαθες στο κεφάλαιο 7.

variables.tf

```
variable "project" {
  type    = string
  default = "shop"
}

variable "environment" {
  type    = string
  default = "dev"
}
```

Το `main.tf` έχει τους providers και δύο resources. Το `random_pet` του κεφαλαίου 6 δίνει ένα τυχαίο όνομα στον server. Το `local_file` γράφει ένα αρχείο ρυθμίσεων που περιέχει αυτό το όνομα.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "server" {
  length = 2
}

resource "local_file" "config" {
  filename = "${var.project}-${var.environment}.conf"
  content  = "server = ${random_pet.server.id}\n"
}
```

Τα outputs μπαίνουν στο `outputs.tf`.

outputs.tf

```
output "config_path" {
  value       = local_file.config.filename
  description = "Το αρχείο ρυθμίσεων του server"
}

output "server_name" {
  value       = random_pet.server.id
  description = "Το τυχαίο όνομα του server"
}
```

Ένα block `output` έχει ένα label, το όνομα του output. Το όνομα είναι μοναδικό μέσα στον φάκελο. Το μόνο υποχρεωτικό argument είναι το `value`, που δέχεται οποιαδήποτε έκφραση. Εδώ το `config_path` παίρνει το attribute `filename` του `local_file.config` και το `server_name` το `id` του `random_pet.server`, με τις αναφορές του κεφαλαίου 6. Το `description` είναι προαιρετικό και εξηγεί σε ανθρώπους τι είναι η τιμή.

Το χώρισμα σε `main.tf`, `variables.tf` και `outputs.tf` είναι σύμβαση. Το Terraform διαβάζει όλα τα αρχεία `.tf` του φακέλου μαζί, όπως είδες στο κεφάλαιο 2. Τα ονόματα βοηθούν τους ανθρώπους να βρίσκουν τι ψάχνουν.

Τα outputs στο plan

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.config will be created
  + resource "local_file" "config" {
      + content              = (known after apply)
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "shop-dev.conf"
      + id                   = (known after apply)
    }

  # random_pet.server will be created
  + resource "random_pet" "server" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + config_path = "shop-dev.conf"
  + server_name = (known after apply)

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Μετά την περίληψη `Plan: 2 to add` το plan έχει μια δεύτερη ενότητα, `Changes to Outputs`. Εκεί φαίνεται τι θα γίνει με κάθε output, με τα ίδια σύμβολα που ξέρεις από τα resources. Το `+` σημαίνει ότι το output είναι καινούργιο.

Το `config_path` έχει ήδη τιμή, `"shop-dev.conf"`. Το `filename` βγαίνει από δύο μεταβλητές, οπότε το Terraform το ξέρει από το plan. Το `server_name` γράφει `(known after apply)`. Το όνομα του `random_pet` θα το διαλέξει ο provider τη στιγμή της δημιουργίας, άρα το ίδιο ισχύει και για κάθε output που το χρησιμοποιεί. Για τον ίδιο λόγο και το `content` του αρχείου είναι `(known after apply)`.

Τα outputs μετά το apply

Από το κεφάλαιο 3 ξέρεις το `-auto-approve`. Θα το χρησιμοποιούμε όταν το plan έχει ήδη φανεί.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.server: Creating...
random_pet.server: Creation complete after 0s [id=tidy-yeti]
local_file.config: Creating...
local_file.config: Creation complete after 0s [id=8cd14b1559455e396a28851faefb07cf95fc3da9]

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

Outputs:

config_path = "shop-dev.conf"
server_name = "tidy-yeti"
```

Μετά το `Apply complete!` το Terraform τυπώνει την ενότητα `Outputs:` με όλα τα outputs και τις τελικές τιμές τους. Το `server_name` έχει πλέον το τυχαίο όνομα που πήρε ο server. Στη δική σου εκτέλεση θα είναι άλλο. Το `description` δεν τυπώνεται πουθενά εδώ. Είναι τεκμηρίωση για όποιον διαβάζει τον κώδικα.

Η εντολή terraform output

Η ενότητα `Outputs:` χάνεται μόλις κυλήσει η οθόνη. Η εντολή `terraform output` ξανατυπώνει τα outputs όποτε τη χρειαστείς, χωρίς plan και χωρίς apply.

TERMINAL

```
terraform output
```



ΕΞΟΔΟΣ

```
config_path = "shop-dev.conf"
server_name = "tidy-yeti"
```

Με όνομα output τυπώνει μόνο αυτό.

TERMINAL

```
terraform output server_name
```

ΕΞΟΔΟΣ

```
"tidy-yeti"
```

Η τιμή βγαίνει σε εισαγωγικά, όπως θα τη γράφαμε σε HCL. Για άνθρωπο αυτό είναι σαφές. Για script είναι πρόβλημα. Δοκίμασε να δώσεις το `config_path` στο `cat`.

TERMINAL

```
cat "$(terraform output config_path)"
```

ΕΞΟΔΟΣ

```
cat: "shop-dev.conf": No such file or directory
```

Το `cat` έψαξε αρχείο που το όνομά του περιέχει και τα εισαγωγικά. Τέτοιο αρχείο δεν υπάρχει. Τη λύση τη δίνει το `-raw`, που τυπώνει τη σκέτη τιμή.

TERMINAL

```
terraform output -raw config_path
cat "$(terraform output -raw config_path)"
```

ΕΞΟΔΟΣ

```
shop-dev.confserver = tidy-yeti
```

Η πρώτη εντολή τύπωσε το όνομα χωρίς εισαγωγικά και η δεύτερη το περιεχόμενο του αρχείου. Πρόσεξε ότι οι δύο τιμές βγήκαν στην ίδια γραμμή. Το `-raw` δεν προσθέτει αλλαγή γραμμής στο τέλος, ώστε η τιμή να περνά αυτούσια σε άλλη εντολή. Στο terminal σου θα δεις το prompt να κολλά δίπλα στην τιμή. Το `-raw` δουλεύει για strings, αριθμούς και bool. Τις σύνθετες τιμές θα τις γνωρίσεις στο κεφάλαιο 9.

Όταν ένα πρόγραμμα χρειάζεται όλα τα outputs μαζί, το `-json` τα δίνει σε μορφή που διαβάζει κάθε γλώσσα.

TERMINAL

```
terraform output -json
```

ΕΞΟΔΟΣ

```
{
  "config_path": {
    "sensitive": false,
    "type": "string",
    "value": "shop-dev.conf"
  },
  "server_name": {
    "sensitive": false,
    "type": "string",
    "value": "tidy-yeti"
  }
}
```

Το αποτέλεσμα είναι ένα αντικείμενο JSON με ένα κλειδί για κάθε output. Κάθε κλειδί έχει την τιμή στο `value`, τον τύπο της στο `type` και το `sensitive`, που θα δεις παρακάτω. Με το `jq`, που είδες στο κεφάλαιο 4, βγάζεις όποιο κομμάτι θέλεις.

TERMINAL

```
terraform output -json | jq -r '.server_name.value'
```

ΕΞΟΔΟΣ

```
tidy-yeti
```

Το `-r` του `jq` αφαιρεί τα εισαγωγικά, όπως το `-raw` του Terraform. Το `-json` μαζί με όνομα, για παράδειγμα `terraform output -json server_name`, τυπώνει μόνο την τιμή εκείνου του output σε JSON.

ΚΡΑΤΑ ΑΥΤΟ

Το `terraform output` σκέτο είναι για ανθρώπους. Το `terraform output -raw ΟΝΟΜΑ` δίνει μία τιμή σε shell script. Το `terraform output -json` δίνει όλα τα outputs σε προγράμματα και στο `jq`.

Τα outputs ζουν στο state

Το `terraform output` δεν διαβάζει τη configuration. Διαβάζει το state. Το apply έγραψε εκεί τις τιμές των outputs, δίπλα στα resources.

TERMINAL

```
jq '.outputs' terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "config_path": {
    "value": "shop-dev.conf",
    "type": "string"
  },
  "server_name": {
    "value": "tidy-yeti",
    "type": "string"
  }
}
```

Αυτό έχει μια συνέπεια που ξαφνιάζει στην αρχή. Πρόσθεσε ένα τρίτο output στο τέλος του `outputs.tf`.

outputs.tf

```
output "greeting" {
  value = "Ο server ${random_pet.server.id} τρέχει στο ${var.environment}"
}
```

Ζήτησέ το πριν από οποιοδήποτε apply.

TERMINAL

```
terraform output greeting
```

ΕΞΟΔΟΣ

```
╷
│ Error: Output "greeting" not found
│
│ The output variable requested could not be found in the state file. If you
│ recently added this to your configuration, be sure to run `terraform
│ apply`, since the state won't be updated with new output variables until
│ that command is run.
╵
```

Το μήνυμα το λέει καθαρά. Το output δεν βρέθηκε στο state και ένα νέο output μπαίνει εκεί μόνο με apply. Δες τι προτείνει το plan.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.server: Refreshing state... [id=tidy-yeti]
local_file.config: Refreshing state... [id=8cd14b1559455e396a28851faefb07cf95fc3da9]

Changes to Outputs:
  + greeting    = "O server tidy-yeti τρέχει στο dev"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Δεν υπάρχει γραμμή `Plan:`, γιατί κανένα resource δεν αλλάζει. Υπάρχει μόνο η ενότητα `Changes to Outputs` με το νέο `greeting`. Η σημείωση από κάτω λέει ότι το apply αυτού του plan θα γράψει τις νέες τιμές στο state χωρίς να αγγίξει πραγματική υποδομή.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
random_pet.server: Refreshing state... [id=tidy-yeti]
local_file.config: Refreshing state... [id=8cd14b1559455e396a28851faefb07cf95fc3da9]

Changes to Outputs:
  + greeting    = "O server tidy-yeti τρέχει στο dev"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

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

Outputs:

config_path = "shop-dev.conf"
greeting = "O server tidy-yeti τρέχει στο dev"
server_name = "tidy-yeti"
```

Η περίληψη λέει `0 added, 0 changed, 0 destroyed`, αλλά τα outputs είναι πλέον τρία. Το `greeting` μπήκε ανάμεσα στα άλλα δύο, γιατί τα outputs τυπώνονται με αλφαβητική σειρά και όχι με τη σειρά των αρχείων. Με τον ίδιο τρόπο ένα output που αλλάζει τιμή εμφανίζεται με `~` και ένα που σβήνεις από τη configuration με `-` και `-> null`.

Ένα output δεν χρειάζεται καν resource. Μια configuration με μόνο variables, locals και outputs δεν χρειάζεται provider. Το `terraform init` και το apply τρέχουν κανονικά και η περίληψη λέει `0 added`. Είναι βολικός τρόπος να δοκιμάζεις εκφράσεις.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά από πού διαβάζει το `terraform output`. Η απάντηση είναι το state. Ένα output που μόλις έγραψες στη configuration δεν υπάρχει για το `terraform output` μέχρι το επόμενο apply. Ένα apply που αλλάζει μόνο outputs ενημερώνει το state χωρίς να αλλάξει κανένα resource.

description, sensitive και depends_on

Εκτός από το `value` ένα output δέχεται μερικά ακόμη arguments.

Το **`description`** δεν επηρεάζει τίποτα στην εκτέλεση. Δεν το τυπώνει ούτε το `terraform output -json`. Αν αλλάξεις μόνο το description, το plan λέει `No changes`. Η αξία του φαίνεται στα modules. Το Terraform Registry και τα εργαλεία τεκμηρίωσης δείχνουν τα descriptions των outputs ως τη λίστα με όσα επιστρέφει το module.

Το **`sensitive = true`** κρύβει την τιμή από την οθόνη. Στο plan γράφεται `(sensitive value)` και στη λίστα του `terraform output` γράφεται `<sensitive>`. Η προστασία όμως είναι μόνο στην οθόνη. Το `terraform output ΟΝΟΜΑ`, το `-raw` και το `-json` τυπώνουν την πραγματική τιμή και το state την κρατά σε απλό κείμενο. Αν ένα output περιέχει τιμή που ο provider σημειώνει ως μυστική, όπως το `result` ενός `random_password`, το Terraform αρνείται να το δεχτεί χωρίς `sensitive = true` και βγάζει το σφάλμα `Output refers to sensitive values`.

```hcl
output "db_password" {
  value     = random_password.db.result
  sensitive = true
}
```

Το κεφάλαιο 24 εξετάζει τα secrets αναλυτικά.

Το **`depends_on`** χρειάζεται σπάνια. Ένα output εξαρτάται αυτόματα από ό,τι αναφέρει η έκφρασή του, όπως ένα resource στο κεφάλαιο 6. Αν η τιμή είναι σωστή μόνο αφού ολοκληρωθεί κάποιο resource που η έκφραση δεν αναφέρει, γράφεις την εξάρτηση ρητά.

```hcl
output "ready" {
  value      = "Οι ρυθμίσεις γράφτηκαν"
  depends_on = [local_file.config]
}
```

Τα outputs έχουν και δεύτερο ρόλο. Τα outputs ενός child module είναι ο μόνος τρόπος να περάσει μια τιμή από αυτό στη configuration που το καλεί. Το κεφάλαιο 15 το δείχνει στην πράξη.

> **ΠΑΓΙΔΑ**
> Ένα output δεν διαβάζεται μέσα στον ίδιο φάκελο. Δεν υπάρχει αναφορά `output.greeting`. Αν τη γράψεις, το Terraform τη διαβάζει σαν resource τύπου `output` και απαντά `A managed resource "output" "greeting" has not been declared`. Αν χρειάζεσαι την ίδια τιμή σε resource και σε output, βάλε την έκφραση σε local και χρησιμοποίησε το local και στα δύο.

Τα locals

Το πρόβλημα της επανάληψης

Ο server χρειάζεται και ένα αρχείο καλωσορίσματος. Πρόσθεσέ το στο τέλος του `main.tf`.



main.tf

```
resource "local_file" "motd" {
  filename = "${var.project}-${var.environment}.motd"
  content  = "Καλώς ήρθες στο ${var.project}-${var.environment}\n"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
```

Η configuration δουλεύει, αλλά η έκφραση `"${var.project}-${var.environment}"` εμφανίζεται πλέον τρεις φορές. Αν αύριο το πρόθεμα πρέπει να γίνει `shop.dev` αντί για `shop-dev`, θα το αλλάξεις σε τρία σημεία. Σε μια πραγματική configuration τα σημεία είναι δεκάδες και κάποιο θα ξεχαστεί.

Η πρώτη ιδέα είναι μια μεταβλητή με default που χρησιμοποιεί τις άλλες δύο. Το κεφάλαιο 7 είπε ότι αυτό δεν επιτρέπεται. Δες πώς απαντά το Terraform, σε χωριστό αρχείο που θα σβήσεις μετά.

prefix.tf

```
variable "name_prefix" {
  type    = string
  default = "${var.project}-${var.environment}"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Variables not allowed
│
│   on prefix.tf line 3, in variable "name_prefix":
│    3:   default = "${var.project}-${var.environment}"
│
│ Variables may not be used here.
╵

╷
│ Error: Variables not allowed
│
│   on prefix.tf line 3, in variable "name_prefix":
│    3:   default = "${var.project}-${var.environment}"
│
│ Variables may not be used here.
╵
```

Το Terraform αρνείται με `Variables not allowed`, μία φορά για κάθε αναφορά στο `default`. Μια μεταβλητή είναι είσοδος. Η τιμή της πρέπει να είναι γνωστή πριν υπολογιστεί οτιδήποτε άλλο, οπότε το `default` δέχεται μόνο σταθερή τιμή. Σβήσε το αρχείο.

TERMINAL

```
rm prefix.tf
```

Το block locals

Το εργαλείο για τιμές που υπολογίζονται είναι το block `locals`. Γράψε το `locals.tf`.

locals.tf

```
locals {
  name_prefix = "${var.project}-${var.environment}"
}
```

Το block `locals` δεν έχει label. Μέσα του γράφεις όσες αναθέσεις `όνομα = έκφραση` θέλεις. Κάθε ανάθεση ορίζει ένα local. Στην υπόλοιπη configuration το διαβάζεις ως `local.name_prefix`. Το block λέγεται `locals` στον πληθυντικό, η αναφορά όμως ξεκινά με `local` στον ενικό.

Άλλαξε το `main.tf` ώστε να χρησιμοποιεί το local.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

```
resource "random_pet" "server" {
  length = 2
}

resource "local_file" "config" {
  filename = "${local.name_prefix}.conf"
  content  = "server = ${random_pet.server.id}\n"
}

resource "local_file" "motd" {
  filename = "${local.name_prefix}.motd"
  content  = "Καλώς ήρθες στο ${local.name_prefix}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.server: Refreshing state... [id=tidy-yeti]
local_file.motd: Refreshing state... [id=fd537ec3853bf8e60f1fd5607cd7b21c7f2f5914]
local_file.config: Refreshing state... [id=8cd14b1559455e396a28851faefb07cf95fc3da9]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το plan λέει `No changes`. Το local είναι απλώς ένα όνομα για μια έκφραση. Το Terraform υπολογίζει την έκφραση, βάζει το αποτέλεσμα στη θέση της αναφοράς και συγκρίνει τις τελικές τιμές των arguments με το state. Οι τιμές είναι ίδιες με πριν, άρα δεν υπάρχει τίποτα να αλλάξει. Αυτό κάνει τα locals ασφαλές εργαλείο για τακτοποίηση. Ένα refactoring που δίνει `No changes` έχει αποδείξει ότι δεν άλλαξε τίποτα.

Τα locals δεν γράφονται ούτε στο state. Δες τα κλειδιά του αρχείου.

TERMINAL

```
jq 'keys' terraform.tfstate
```

ΕΞΟΔΟΣ

```
[
  "check_results",
  "lineage",
  "outputs",
  "resources",
  "serial",
  "terraform_version",
  "version"
]
```

Υπάρχουν `outputs` και `resources`, αλλά κανένα κλειδί για locals. Κάθε εκτέλεση τα υπολογίζει από την αρχή.

Locals από locals, resources και αριθμούς

Ένα local μπορεί να χρησιμοποιεί μεταβλητές, attributes από resources και άλλα locals. Πρόσθεσε μια μεταβλητή για τη θύρα βάσης.

variables.tf

```
variable "base_port" {
  type    = number
  default = 8000
}
```

Μετά γράψε ξανά το `locals.tf` με τρία ακόμη locals.

locals.tf

```
locals {
  name_prefix = "${var.project}-${var.environment}"
  header      = "# ${local.name_prefix}, διαχειρίζεται το Terraform\n"
  app_port    = var.base_port + 80
  app_url     = "http://${random_pet.server.id}:${local.app_port}"
}
```

Το `header` είναι μια κοινή πρώτη γραμμή για όλα τα αρχεία και χρησιμοποιεί το `name_prefix`. Το `app_port` κάνει αριθμητική με μια μεταβλητή. Το `app_url` συνδέει ένα attribute του `random_pet` με άλλο local. Η σειρά των γραμμών δεν παίζει ρόλο. Το Terraform βρίσκει μόνο του τι πρέπει να υπολογιστεί πρώτο από τις αναφορές, όπως έκανε με τα resources στο κεφάλαιο 6.

Το `header` είναι ένα μικρό δείγμα από τα κοινά στοιχεία που μοιράζονται πολλά resources. Στο κεφάλαιο 17 το ίδιο μοτίβο δίνει ένα κοινό πρόθεμα στα ονόματα δύο buckets του AWS.

Χρησιμοποίησε τα νέα locals στα δύο αρχεία. Στο `main.tf` άλλαξαν μόνο τα δύο `content`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "server" {
  length = 2
}

resource "local_file" "config" {
  filename = "${local.name_prefix}.conf"
  content  = "${local.header}url = ${local.app_url}\n"
}

resource "local_file" "motd" {
  filename = "${local.name_prefix}.motd"
```

```
  content  = "${local.header}Καλώς ήρθες στο ${local.name_prefix}\n"
}
```

Πρόσθεσε και ένα output για το `app_url`, ώστε η τιμή του να φαίνεται μετά το apply.

outputs.tf

```
output "app_url" {
  value = local.app_url
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
  # local_file.config must be replaced
…
  # local_file.motd must be replaced
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
Changes to Outputs:
  + app_url     = "http://tidy-yeti:8080"
…
Apply complete! Resources: 2 added, 0 changed, 2 destroyed.
…
```

Τα δύο αρχεία αντικαθίστανται, γιατί άλλαξε το `content` τους και ο provider `local` αντικαθιστά το αρχείο σε κάθε τέτοια αλλαγή. Το νέο output εμφανίζεται στο `Changes to Outputs` με `+`.

TERMINAL

```
cat shop-dev.conf
```

ΕΞΟΔΟΣ

```
# shop-dev, διαχειρίζεται το Terraform
url = http://tidy-yeti:8080
```

Η πρώτη γραμμή ήρθε από το `header` και η θύρα `8080` από την πράξη `8000 + 80`. Ο αριθμός μπήκε στο string χωρίς καμία μετατροπή από σένα.

Οι κανόνες των locals

- Μπορείς να έχεις πολλά blocks `locals`, στο ίδιο ή σε διαφορετικά αρχεία. Κάθε όνομα όμως ορίζεται μία φορά σε όλο τον φάκελο. Ένα διπλό όνομα δίνει το σφάλμα `Duplicate local value definition`.
- Δύο locals που χρειάζεται το καθένα το άλλο δίνουν σφάλμα `Cycle`, γιατί κανένα δεν μπορεί να υπολογιστεί πρώτο.
- Ένα local που εξαρτάται από attribute άγνωστο στο plan είναι κι αυτό `(known after apply)` μέχρι το apply.

- Τα locals δεν εμφανίζονται στο plan, στο state ή στο `terraform output`. Για να δεις την τιμή ενός local, βάλε το σε output, όπως το `app_url`.

Το πιο συνηθισμένο λάθος είναι ο πληθυντικός στην αναφορά. Δοκίμασέ το σε προσωρινό αρχείο.

typo.tf

```
output "prefix" {
  value = locals.name_prefix
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Reference to undeclared resource
│
│   on typo.tf line 2, in output "prefix":
│    2:   value = locals.name_prefix
│
│ A managed resource "locals" "name_prefix" has not been declared in the root
│ module.
╵
```

TERMINAL

```
rm typo.tf
```

Το Terraform διάβασε το `locals.name_prefix` σαν αναφορά σε resource τύπου `locals` με όνομα `name_prefix`. Τέτοιο resource δεν υπάρχει. Όταν δεις `A managed resource "locals"` σε μήνυμα σφάλματος, ψάξε για ένα `s` που περισσεύει.

ΠΑΓΙΔΑ

Μη μετατρέπεις κάθε τιμή σε local. Ένα local που χρησιμοποιείται μία φορά αναγκάζει τον αναγνώστη να ψάξει τι σημαίνει σε άλλο σημείο του κώδικα. Τα locals αξίζουν όταν μια έκφραση επαναλαμβάνεται ή όταν ένα όνομα εξηγεί καλύτερα έναν σύνθετο υπολογισμό.

Πότε local και πότε variable

Οι δύο έννοιες μοιάζουν, γιατί και οι δύο δίνουν όνομα σε μια τιμή. Η διαφορά είναι ποιος ελέγχει την τιμή.

	Variable	Local
Ρόλος	Είσοδος από όποιον τρέχει τη configuration	Εσωτερικός υπολογισμός
Ορισμός	Block `variable "όνομα"`	Ανάθεση μέσα σε block `locals`
Αναφορά	`var.όνομα`	`local.όνομα`

	Variable	Local
Τιμή απ' έξω	`-var`, αρχεία `.tfvars`, `TF_VAR_`	Ποτέ
Χρήση άλλων τιμών	Όχι στο `default`	Ναι

Ένα local δεν αλλάζει από τη γραμμή εντολών. Αν δοκιμάσεις, το Terraform ψάχνει μεταβλητή με αυτό το όνομα.

TERMINAL

```
terraform plan -var name_prefix=test
```

ΕΞΟΔΟΣ

```
Error: Value for undeclared variable

A variable named "name_prefix" was assigned on the command line, but the
root module does not declare a variable of that name. To use this value,
add a "variable" block to the configuration.
```

Το σφάλμα `Value for undeclared variable` λέει ότι δεν υπάρχει block `variable` με όνομα `name_prefix`. Το local με το ίδιο όνομα δεν μετράει.

ΠΑΓΙΔΑ

Όπως είδες στο κεφάλαιο 7, μόνο το `-var` σταματά με σφάλμα. Μια τιμή `name_prefix` σε `terraform.tfvars` δίνει απλή προειδοποίηση και αγνοείται. Ένα `TF_VAR_name_prefix` στο περιβάλλον αγνοείται χωρίς κανένα μήνυμα. Αν μια τιμή που δίνεις δεν φαίνεται να επηρεάζει τίποτα, έλεγξε ότι αντιστοιχεί σε block `variable` και όχι σε local.

Ο κανόνας για την επιλογή είναι απλός. Αν η τιμή διαφέρει από περιβάλλον σε περιβάλλον ή πρέπει να τη διαλέγει όποιος τρέχει τη configuration, θέλεις variable. Αν η τιμή βγαίνει από άλλες τιμές ή είναι εσωτερική λεπτομέρεια που δεν θέλεις να αλλάζει κανείς, θέλεις local. Εδώ το `project` και το `environment` είναι variables, ενώ το `name_prefix` είναι local, γιατί βγαίνει πάντα από αυτά τα δύο.

Outputs και destroy

Το destroy αφαιρεί και τα outputs από το state.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Changes to Outputs:
  - app_url     = "http://tidy-yeti:8080" -> null
  - config_path = "shop-dev.conf" -> null
  - greeting    = "Ο server tidy-yeti τρέχει στο dev" -> null
  - server_name = "tidy-yeti" -> null
…
Destroy complete! Resources: 3 destroyed.
```

Το plan του destroy έχει κι αυτό ενότητα `Changes to Outputs`. Κάθε output γίνεται `null`, ακόμη κι ένα output με σταθερή τιμή. Το destroy αδειάζει το state και από resources και από outputs.

TERMINAL

```
terraform output
```

ΕΞΟΔΟΣ

```
…
╷ Warning: No outputs found
…
```

Το state δεν έχει πια outputs και το `terraform output` το λέει με προειδοποίηση.

Σύνοψη

Θέλεις	Γράφεις
Μια τιμή να φαίνεται μετά το apply	Block `output` με `value`
Να ξαναδείς τα outputs	`terraform output`
Μία τιμή σε shell script	`terraform output -raw ONOMA`
Όλα τα outputs σε πρόγραμμα	`terraform output -json`
Να κρύψεις μια τιμή από την οθόνη	`sensitive = true`
Όνομα για υπολογισμό που επαναλαμβάνεται	Block `locals`, αναφορά `local.όνομα`
Τιμή που αλλάζει όποιος τρέχει τη configuration	Block `variable`, αναφορά `var.όνομα`

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Πρόσεξε τις λεπτομέρειες που ελέγχει συχνά η εξέταση. Το block είναι `locals` και η αναφορά `local.όνομα`. Τα locals δεν παίρνουν τιμή από `-var` ή `.tfvars`. Το `default` μιας μεταβλητής δεν μπορεί να αναφέρει άλλες μεταβλητές. Το `-raw` τυπώνει μία τιμή χωρίς εισαγωγικά και το `-json` όλα τα outputs σε JSON. Ένα output με `sensitive = true` κρύβεται από την οθόνη αλλά μένει στο state.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch08`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από αντίγραφο άλλου. Όπου χρειάζεται provider, δήλωσέ τον στο block `terraform` με την έκδοση του κεφαλαίου. Τρέξε `terraform init` πριν από το πρώτο apply.

ΑΣΚΗΣΗ 8.1 Το πρώτο output

Γράψε configuration με ένα resource `local_file` με όνομα `note`, που φτιάχνει το `note.txt` με περιεχόμενο `Σημειώσεις έργου\n`. Πρόσθεσε output `note_path` με το `filename` του αρχείου και ένα `description`. Κάνε apply με `-auto-approve`. Το τέλος της εξόδου πρέπει να είναι το παρακάτω.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Outputs:

note_path = "note.txt"
```

ΑΣΚΗΣΗ 8.2 Μια τιμή για script

Αντέγραψε τα αρχεία της άσκησης 8.1 σε νέο φάκελο, χωρίς το state. Τρέξε `terraform init` και apply. Με μία εντολή shell αντέγραψε το αρχείο που δείχνει το output `note_path` στο `backup.txt`, χωρίς να γράψεις το όνομα του αρχείου με το χέρι. Μετά τύπωσε το `backup.txt`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Σημειώσεις έργου
```

ΑΣΚΗΣΗ 8.3 Ένα output, τρεις μορφές

Γράψε configuration χωρίς resources και χωρίς providers, με ένα μόνο output `city` και τιμή `Θεσσαλονίκη`. Κάνε apply. Μετά τύπωσε το `city` τρεις φορές, με `terraform output city`, με `-raw` και με `-json`. Βάλε ένα `echo` μετά το `-raw`, ώστε κάθε τιμή να βγει σε δική της γραμμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
"Θεσσαλονίκη"
Θεσσαλονίκη
"Θεσσαλονίκη"
```

ΑΣΚΗΣΗ 8.4 Αλλαγή μόνο σε outputs

Αντέγραψε τα αρχεία της άσκησης 8.1 σε νέο φάκελο, τρέξε `terraform init` και apply. Μετά κάνε δύο αλλαγές. Άλλαξε το κείμενο του `description` στο `note_path` και πρόσθεσε output `note_id` με το `id` του `local_file.note`. Κάνε apply ξανά. Ποια από τις δύο αλλαγές εμφανίζεται στο plan και τι λέει η περίληψη;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Changes to Outputs:
  + note_id   = "429b567c780425366f0afca15cfaea938cc9f9d6"
…
You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
…
```

ΑΣΚΗΣΗ 8.5 Από την επανάληψη σε local

Ξεκίνα από το παρακάτω `main.tf`. Τρέξε `terraform init` και apply. Μετά βάλε την έκφραση που επαναλαμβάνεται σε ένα local με όνομα `prefix` και χρησιμοποίησέ το στα τρία σημεία. Απόδειξε με `terraform plan` ότι η αλλαγή δεν αγγίζει κανένα resource.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "team" {
  type    = string
  default = "data"
}

variable "stage" {
  type    = string
  default = "test"
}

resource "local_file" "input" {
  filename = "${var.team}-${var.stage}-input.txt"
  content  = "είσοδος\n"
}

resource "local_file" "report" {
  filename = "${var.team}-${var.stage}-report.txt"
  content  = "αναφορά για ${var.team}-${var.stage}\n"
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
No changes. Your infrastructure matches the configuration.
…
```

ΑΣΚΗΣΗ 8.6 Local ή variable

Ξεκίνα από το παρακάτω `main.tf`. Τρέξε `terraform init` και μετά `terraform apply -auto-approve -var environment=prod`. Διάβασε το σφάλμα. Άλλαξε τη configuration ώστε η εντολή να πετύχει, ενώ χωρίς `-var` το περιβάλλον να παραμένει `dev`. Στο τέλος τύπωσε το output `env_file` με `-raw`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

locals {
  environment = "dev"
}

resource "local_file" "env" {
  filename = "${local.environment}.env"
  content  = "ENVIRONMENT=${local.environment}\n"
}

output "env_file" {
  value = local_file.env.filename
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
prod.env
```

ΑΣΚΗΣΗ 8.7 Αριθμοί σε locals

Γράψε configuration χωρίς resources με δύο μεταβλητές τύπου `number`, τη `servers` με default `3` και τη `disk_gb` με default `20`. Υπολόγισε σε locals τον συνολικό δίσκο και το μηνιαίο κόστος, με 2 ευρώ ανά GB. Ο πολλαπλασιασμός γράφεται με `*`, όπως στις περισσότερες γλώσσες. Βγάλε ένα output `summary` με κείμενο της μορφής `3 servers, 60 GB, 120 ευρώ`. Κάνε apply με `-var servers=5` και τύπωσε το `summary` με `-raw`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
5 servers, 100 GB, 200 ευρώ
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 8.1

Ένα script που κάνει deployment χρειάζεται την τιμή του output `bucket_name` χωρίς εισαγωγικά, για να τη δώσει σε άλλη εντολή. Ποια εντολή ταιριάζει;

a. `terraform output bucket_name`
b. `terraform show bucket_name`
c. `terraform output -raw bucket_name`
d. `terraform output -json`

ΕΡΩΤΗΣΗ 8.2

Ένα block `locals` ορίζει το `name_prefix`. Πώς το χρησιμοποιείς μέσα σε ένα resource;

a. `locals.name_prefix`
b. `local.name_prefix`
c. `var.name_prefix`
d. `locals["name_prefix"]`

ΕΡΩΤΗΣΗ 8.3

Προσθέτεις ένα νέο block `output "endpoint"` στη configuration και αμέσως τρέχεις `terraform output endpoint`, χωρίς apply. Τι γίνεται;

a. Τυπώνεται η τιμή, που το Terraform υπολογίζει από τη configuration.
b. Τυπώνεται `(known after apply)`.
c. Τυπώνεται κενό string.
d. Η εντολή αποτυγχάνει με μήνυμα ότι το output δεν βρέθηκε, γιατί διαβάζει τα outputs από το state.

ΕΡΩΤΗΣΗ 8.4

Ποιες δύο προτάσεις για τα locals είναι σωστές; Επίλεξε δύο.

a. Παίρνουν τιμή από τη γραμμή εντολών με `-var`.
b. Μπορούν να χρησιμοποιούν input variables, attributes από resources και άλλα locals.
c. Γράφονται στο state και εμφανίζονται στο `terraform output`.
d. Ένας φάκελος μπορεί να έχει πολλά blocks `locals`, αρκεί κάθε όνομα να ορίζεται μία φορά.
e. Κάθε block `locals` ορίζει μία μόνο τιμή.

ΕΡΩΤΗΣΗ 8.5

Το plan δείχνει μόνο την ενότητα `Changes to Outputs` και καμία αλλαγή σε resources. Τι κάνει το `terraform apply`;

a. Γράφει τις νέες τιμές των outputs στο state χωρίς να αλλάξει υποδομή.
b. Αντικαθιστά τα resources που αναφέρουν τα outputs.
c. Δεν κάνει τίποτα, γιατί τα outputs υπολογίζονται από την αρχή σε κάθε `terraform output`.
d. Αποτυγχάνει, γιατί ένα apply χρειάζεται τουλάχιστον μία αλλαγή σε resource.

ΕΡΩΤΗΣΗ 8.6

Θέλεις το πρόθεμα `"${var.project}-${var.environment}"` σε πολλά resources και δεν θέλεις να μπορεί να το αλλάξει κανείς από τη γραμμή εντολών. Τι γράφεις;

a. Ένα block `variable "prefix"` με αυτή την έκφραση στο `default`.
b. Ένα block `output "prefix"` και αναφορές `output.prefix` στα resources.
c. Ένα block `locals` με `prefix` ίσο με αυτή την έκφραση.
d. Μια γραμμή `prefix` με αυτή την έκφραση στο `terraform.tfvars`.



ΚΕΦΑΛΑΙΟ 9

Τύποι και δομές δεδομένων

Κάθε τιμή στο Terraform έχει τύπο. Σε αυτό το κεφάλαιο γνωρίζεις όλους τους τύπους, μαθαίνεις πότε το Terraform μετατρέπει μόνο του μια τιμή και πώς διαβάζεις ένα στοιχείο μέσα από μια δομή. Εργαλείο σου είναι το `terraform console`, όπου δοκιμάζεις κάθε έκφραση πριν τη γράψεις σε αρχείο.

Το terraform console

Μέχρι τώρα, για να δεις την τιμή μιας έκφρασης, έγραφες ένα output και έτρεχες apply. Το `terraform console` είναι πιο γρήγορο. Γράφεις μια έκφραση, πατάς Enter και βλέπεις αμέσως την τιμή της.

Φτιάξε έναν άδειο φάκελο για τις πρώτες δοκιμές και μπες μέσα.

```
TERMINAL

mkdir explore
cd explore
```

Στο terminal σου ανοίγεις το console με την εντολή του.

```
TERMINAL

terraform console
```

Η γραμμή εντολών γίνεται `>` και το console περιμένει έκφραση. Για να βγεις γράφεις `exit` ή πατάς Ctrl D. Στο βιβλίο οι εκφράσεις του console φαίνονται σε πλαίσια με την ετικέτα TERRAFORM CONSOLE. Η γραμμή με το `>` είναι όσα πληκτρολογείς και από κάτω είναι η απάντηση.

TERRAFORM CONSOLE

```
> "Γεια σου"
"Γεια σου"
> 8080
8080
> true
true
```

Μια απλή τιμή επιστρέφει τον εαυτό της.

Το console δεν αλλάζει τίποτα. Δεν φτιάχνει plan, δεν κάνει apply και δεν γράφει state. Διαβάζει όμως τη configuration του φακέλου όπου το ανοίγεις, τις τιμές των μεταβλητών και το state, αν υπάρχει. Έτσι μπορείς να ρωτήσεις την τιμή μιας μεταβλητής ή ενός attribute ενός resource που έχεις φτιάξει.

Μια έκφραση περνά και από pipe, χωρίς διαδραστική συνεδρία.

TERMINAL

```
echo 'type("Γεια σου")' | terraform console
```

ΕΞΟΔΟΣ

```
string
```

Η απάντηση `string` είναι ο τύπος της τιμής. Η συνάρτηση `type` υπάρχει μόνο στο console. Αν τη γράψεις σε αρχείο `.tf`, το Terraform απαντά ότι δεν υπάρχει συνάρτηση με αυτό το όνομα.

Οι primitive τύποι

Οι απλοί τύποι λέγονται **primitive** και είναι τρεις.

- `string`, κείμενο σε διπλά εισαγωγικά, για παράδειγμα `"web"`.
- `number`, αριθμός. Ο ίδιος τύπος καλύπτει ακέραιους και δεκαδικούς, όπως το `8080` και το `0.5`.
- `bool`, λογική τιμή, `true` ή `false`.

TERRAFORM CONSOLE

```
> type("web")
string
> type(8080)
number
> type(0.5)
number
> type(false)
bool
```

Υπάρχει και η ειδική τιμή `null`, που σημαίνει απουσία τιμής. Τη βλέπεις αναλυτικά πιο κάτω.

Συλλογές, list, set και map

Μια **συλλογή** (collection type) κρατά όσες τιμές θέλεις, όλες του ίδιου τύπου. Ο τύπος των στοιχείων γράφεται σε παρένθεση. Το `list(string)` είναι λίστα από strings και το `map(number)` είναι map με αριθμούς για τιμές.

- `list(...)` είναι μια σειρά τιμών. Κρατά τη σειρά που έγραψες, επιτρέπει διπλότυπα και κάθε στοιχείο έχει index που ξεκινά από το 0.
- `set(...)` είναι ένα σύνολο μοναδικών τιμών. Δεν έχει διπλότυπα, δεν κρατά σειρά και δεν έχει index.
- `map(...)` αντιστοιχίζει κλειδιά σε τιμές. Τα κλειδιά είναι πάντα strings και όλες οι τιμές έχουν τον ίδιο τύπο.

Οι αγκύλες `[...]` και τα άγκιστρα `{...}` δεν φτιάχνουν απευθείας list ή map, αλλά τους δομικούς τύπους της επόμενης ενότητας. Για συλλογή στο console χρησιμοποιείς τις συναρτήσεις μετατροπής `tolist`, `toset` και `tomap`.

TERRAFORM CONSOLE

```
> tolist(["web", "db", "web"])
tolist([
  "web",
  "db",
  "web",
])
> toset(["web", "db", "web"])
toset([
  "db",
  "web",
])
> tomap({ http = 80, https = 443 })
tomap({
  "http" = 80
  "https" = 443
})
```

Το console γράφει το είδος της συλλογής μπροστά από την τιμή. Η λίστα κράτησε και τα τρία στοιχεία με τη σειρά τους. Το set κράτησε δύο, γιατί το δεύτερο `"web"` ήταν διπλότυπο. Τα έδειξε επίσης ταξινομημένα και όχι με τη σειρά που τα έγραψες. Στο map κάθε κλειδί τυπώθηκε σε εισαγωγικά, γιατί τα κλειδιά είναι strings.

Η συνάρτηση `length` μετρά τα στοιχεία μιας συλλογής. Σε string μετρά χαρακτήρες.

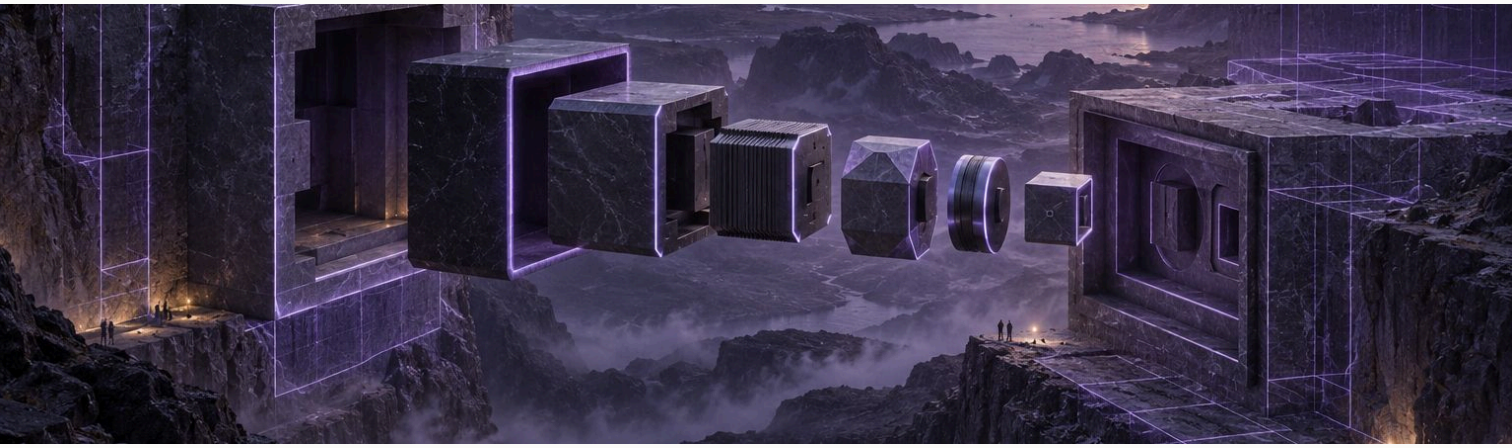
TERRAFORM CONSOLE

```
> length(tolist(["web", "db", "web"]))
3
> length(toset(["web", "db", "web"]))
2
> length("Terraform")
9
```

> **ΠΑΓΙΔΑ**
> Μη βασίζεσαι στη σειρά που δείχνει ένα set. Το set ως έννοια δεν έχει σειρά. Αν η σειρά μετρά, θέλεις list.

Δομικοί τύποι, tuple και object

Οι **δομικοί τύποι** (structural types) επιτρέπουν διαφορετικό τύπο σε κάθε στοιχείο. Σε αντάλλαγμα έχουν σταθερό σχήμα.



- `tuple([...])` είναι σειρά με σταθερό αριθμό θέσεων και δικό της τύπο σε κάθε θέση. Το `tuple([string, number, bool])` έχει ακριβώς τρία στοιχεία, πρώτα ένα string, μετά ένα number και στο τέλος ένα bool.
- `object({...})` έχει σταθερά ονόματα attributes με δικό του τύπο το καθένα. Το `object({ name = string, port = number })` έχει ακριβώς ένα `name` που είναι string και ένα `port` που είναι number.

Οι αγκύλες και τα άγκιστρα φτιάχνουν ακριβώς αυτούς τους τύπους.

TERRAFORM CONSOLE

```
> type(["v1", 3, true])
tuple([
    string,
    number,
    bool,
])
> type({ name = "api", port = 8080 })
object({
    name: string,
    port: number,
})
```

Αυτό που έμοιαζε με λίστα είναι tuple με τρεις θέσεις και αυτό που έμοιαζε με map είναι object με δύο attributes. Το console γράφει `name: string`, ενώ στον κώδικα γράφεις `name = string`.

Τύπος	Τύπος στοιχείων	Σχήμα	Πρόσβαση
`list(T)`	ίδιος για όλα	όσα στοιχεία θέλεις, με σειρά	`[0]`
`set(T)`	ίδιος για όλα	μοναδικά, χωρίς σειρά	καμία
`map(T)`	ίδιος για όλα	όσα κλειδιά θέλεις	`["k"]` ή `.k`
`tuple([...])`	δικός ανά θέση	σταθερές θέσεις	`[0]`
`object({...})`	δικός ανά attribute	σταθερά attributes	`.name` ή `["name"]`

ΚΡΑΤΑ ΑΥΤΟ

Οι συλλογές έχουν έναν τύπο για όλα τα στοιχεία. Οι δομικοί τύποι έχουν σταθερό σχήμα και δικό τους τύπο σε κάθε θέση ή attribute. Για μια ομάδα διαφορετικών ρυθμίσεων, όπως ένα όνομα, ένα port και ένα bool, ο σωστός τύπος είναι object.

Αυτόματες μετατροπές

Στο κεφάλαιο 7 είδες ότι μια μεταβλητή `number` δέχεται το string `"3"`. Γενικά, όταν μια τιμή δεν έχει τον τύπο που χρειάζεται, το Terraform προσπαθεί να τη μετατρέψει μόνο του. Οι κανόνες είναι λίγοι.

- Ένα number ή ένα bool γίνεται πάντα string. Το `8080` γίνεται `"8080"`.
- Ένα string γίνεται number μόνο αν γράφει αριθμό. Το `"443"` γίνεται, το `"http"` όχι.
- Ένα string γίνεται bool όταν είναι `"true"` ή `"false"`. Το `"yes"` δεν γίνεται.
- Ένα tuple γίνεται list και ένα object γίνεται map, αν όλες οι τιμές μπορούν να πάρουν έναν κοινό τύπο.
- Μια list γίνεται set και χάνει τη σειρά της και τα διπλότυπα.

Οι συναρτήσεις `tostring`, `tonumber`, `tolist`, `toset` και `tomap` κάνουν ρητά τις ίδιες μετατροπές.

TERRAFORM CONSOLE

```
> tonumber("443")
443
> tostring(8080)
"8080"
> tolist([80, "http"])
tolist([
  "80",
  "http",
])
> tomap({ name = "api", port = 8080 })
tomap({
  "name" = "api"
  "port" = "8080"
})
```

Το `tolist` πήρε ένα number και ένα string. Ο μόνος κοινός τύπος είναι το string, οπότε το `80` έγινε `"80"`. Το ίδιο έπαθε το `8080` μέσα στο map. Η μετατροπή αποτυγχάνει όταν η τιμή δεν χωρά στον τύπο ή όταν δεν υπάρχει κοινός τύπος.

TERRAFORM CONSOLE

```
> tonumber("http")
│ Error: Invalid function argument
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Invalid value for "v" parameter: cannot convert "http" to number; given
│ string must be a decimal representation of a number.

> tonumber(true)
│ Error: Invalid function argument
```

```
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Invalid value for "v" parameter: cannot convert bool to number.

> tolist(["web", ["db"]])

│ Error: Invalid function argument
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Invalid value for "v" parameter: cannot convert tuple to list of any single
│ type.
```

Το `"http"` δεν γράφει αριθμό. Ένα bool δεν γίνεται ποτέ number. Ένα string και μια λίστα δεν έχουν κοινό τύπο, οπότε το tuple δεν γίνεται list. Η γραμμή `(source code not available)` σημαίνει απλώς ότι η έκφραση δεν προέρχεται από αρχείο.



Σχήμα 9.1 · Οι μετατροπές ανάμεσα στους σύνθετους τύπους. Αριστερά οι δομικοί τύποι, δεξιά οι συλλογές. Τα βέλη δείχνουν προς τα πού γίνεται μετατροπή και πότε.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση δίνει συχνά μια τιμή και ρωτά αν ταιριάζει σε έναν τύπο. Ένα `"443"` περνά σε μεταβλητή τύπου `number`. Ένα `[80, "http"]` περνά σε `list(string)` ως `["80", "http"]`. Ένα `"http"` ή ένα `true` δεν περνούν σε `number`.

Μεταβλητές με σύνθετους τύπους

Το argument `type` του block `variable`, που είδες στο κεφάλαιο 7, δέχεται κάθε τύπο αυτού του κεφαλαίου. Όποια τιμή κι αν δώσει κάποιος, θα μετατραπεί στον τύπο της μεταβλητής ή θα απορριφθεί πριν γίνει οτιδήποτε. Γύρνα στον φάκελο του κεφαλαίου και φτιάξε τον φάκελο `team`.

TERMINAL

```
cd ..
mkdir team
cd team
```

variables.tf

```
variable "members" {
  type = list(string)
}

variable "tags" {
  type = set(string)
}

variable "ports" {
  type = map(number)
}

variable "owner" {
  type = object({
    name  = string
    email = string
    admin = bool
  })
}
```

Κάθε γραμμή μέσα στο `object({...})` είναι ένα attribute και ο τύπος του. Οι τιμές μπαίνουν στο `terraform.tfvars`, που το Terraform διαβάζει μόνο του.

terraform.tfvars

```
members = ["anna", "nikos", "anna"]
tags    = ["web", "db", "web"]
ports = {
  http  = 80
  https = "443"
}
owner = {
  name  = "Άννα"
  email = "anna@example.com"
  admin = "true"
}
```

Το `https` και το `admin` δόθηκαν ως strings, ενώ η μεταβλητή τα θέλει number και bool. Η configuration δεν χρησιμοποιεί κανέναν provider, οπότε το console δουλεύει χωρίς init.

TERRAFORM CONSOLE

```
> var.members
tolist([
  "anna",
  "nikos",
  "anna",
])
> var.tags
toset([
  "db",
  "web",
])
> var.ports
tomap({
  "http" = 80
  "https" = 443
```

```
})
> var.owner
{
  "admin" = true
  "email" = "anna@example.com"
  "name" = "Άννα"
}
```

Η λίστα κράτησε τρία στοιχεία μαζί με το διπλό `"anna"`. Το set κράτησε δύο. Το `443` τυπώθηκε χωρίς εισαγωγικά, άρα έγινε number. Το `admin` έγινε το bool `true`. Το object τυπώνεται χωρίς πρόθεμα, με τα attributes σε αλφαβητική σειρά και με τον τύπο που δήλωσες.

TERRAFORM CONSOLE

```
> type(var.owner)
object({
    admin: bool,
    email: string,
    name: string,
})
```

Πρόσθεσε outputs που δείχνουν τις τιμές.

outputs.tf

```
output "members" {
  value = var.members
}

output "tags" {
  value = var.tags
}

output "owner_admin" {
  value = var.owner.admin
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Changes to Outputs:
  + members     = [
      + "anna",
      + "nikos",
      + "anna",
    ]
  + owner_admin = true
  + tags        = [
      + "db",
      + "web",
    ]

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.
```

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

Outputs:

members = tolist([
  "anna",
  "nikos",
  "anna",
])
owner_admin = true
tags = toset([
  "db",
  "web",
])
```

Δεν υπάρχουν resources, οπότε το apply γράφει μόνο τα outputs στο state. Στο τέλος το `tags` τυπώνεται ως `toset([...])` με δύο στοιχεία και το `owner_admin` ως `true` χωρίς εισαγωγικά.

Το `-raw` του κεφαλαίου 8 δουλεύει μόνο για απλές τιμές.

ΤΕΡΜΙΝΑΛ

```
terraform output -raw tags
```

ΕΞΟΔΟΣ

```
╷
│ Error: Unsupported value for raw output
│
│ The -raw option only supports strings, numbers, and boolean values, but
│ output value "tags" is set of string.
│
│ Use the -json option for machine-readable representations of output values
│ that have complex types.
╵
```

Το μήνυμα λέει ότι το `tags` είναι `set of string` και προτείνει το `-json`. Στο JSON δεν υπάρχει set, οπότε γίνεται πίνακας.

ΤΕΡΜΙΝΑΛ

```
terraform output -json tags
```

ΕΞΟΔΟΣ

```
["db","web"]
```

Μια σύνθετη τιμή στο `-var` γράφεται με σύνταξη HCL, όπως είδες στο κεφάλαιο 7. Δώσε μία που δεν χωρά στον τύπο.

ΤΕΡΜΙΝΑΛ

```
terraform plan -var 'ports={ http = 80, https = "secure" }'
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Invalid value for input variable
│
│   on variables.tf line 9:
│    9: variable "ports" {
│
│ Unsuitable value for var.ports set using -var="ports=...": a number is
│ required.
```

Το plan απέτυχε πριν υπολογίσει οποιαδήποτε αλλαγή. Το σφάλμα δείχνει τη δήλωση της μεταβλητής στο `variables.tf` και την αιτία, `a number is required`. Το `"secure"` δεν γράφει αριθμό. Ο τύπος σταμάτησε τη λάθος τιμή πριν φτάσει σε οποιοδήποτε resource.

Πρόσβαση σε στοιχεία

Για να διαβάσεις ένα στοιχείο γράφεις μετά την τιμή ένα index ή ένα κλειδί.

- Σε list και tuple γράφεις τη θέση σε αγκύλες. Το πρώτο στοιχείο είναι το `[0]`.
- Σε map γράφεις το κλειδί σε αγκύλες, `["https"]`, ή μετά από τελεία, `.https`.
- Σε object γράφεις το όνομα του attribute μετά από τελεία, `.email`. Δουλεύει και σε αγκύλες.

TERRAFORM CONSOLE

```
> var.members[0]
"anna"
> var.members[2]
"anna"
> var.ports["https"]
443
> var.ports.http
80
> var.owner.email
"anna@example.com"
> var.owner["name"]
"Άννα"
```

Οι δύο συντάξεις του map δίνουν το ίδιο αποτέλεσμα. Οι αγκύλες όμως δέχονται και έκφραση. Αν το κλειδί βρίσκεται σε μεταβλητή, γράφεις `var.ports[var.protocol]`. Με τελεία αυτό δεν γίνεται.

Όταν το στοιχείο δεν υπάρχει, το Terraform σταματά με σφάλμα. Δεν επιστρέφει κενή τιμή.

TERRAFORM CONSOLE

```
> var.members[3]
╷
│ Error: Invalid index
│
│   on <console-input> line 1:
│   (source code not available)
│
│ The given key does not identify an element in this collection value: the
│ given index is greater than or equal to the length of the collection.
```

```
> var.ports["ftp"]
╷
│ Error: Invalid index
│
│   on <console-input> line 1:
│   (source code not available)
│
│ The given key does not identify an element in this collection value.
╵
> var.owner.phone
╷
│ Error: Unsupported attribute
│
│   on <console-input> line 1:
│   (source code not available)
│
│ This object does not have an attribute named "phone".
╵
> var.tags[0]
╷
│ Error: Invalid index
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Elements of a set are identified only by their value and don't have any
│ separate index or key to select with, so it's only possible to perform
│ operations across all elements of the set.
╵
```

Η λίστα έχει τρία στοιχεία, στις θέσεις 0 έως 2, οπότε το `[3]` δεν υπάρχει. Το map δεν έχει κλειδί `"ftp"` και το object δεν έχει attribute `phone`. Το set δεν έχει καθόλου index, γιατί τα στοιχεία του αναγνωρίζονται μόνο από την τιμή τους.

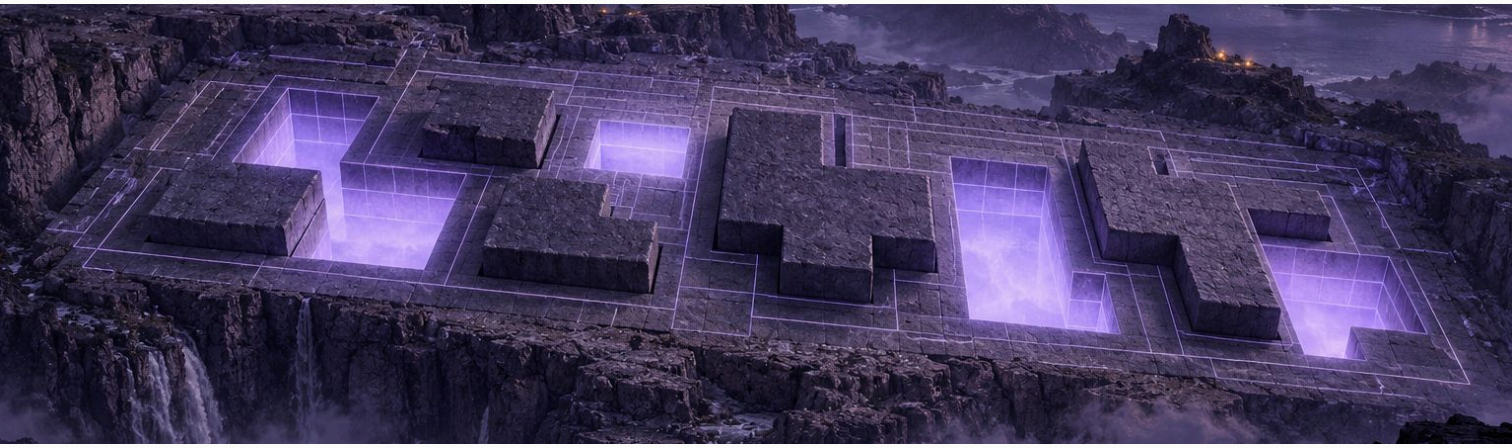
null και any

Η τιμή `null` σημαίνει «καμία τιμή». Όταν ένα argument ενός resource παίρνει `null`, το Terraform το αντιμετωπίζει σαν να μην το έγραψες καθόλου. Αν το argument είναι προαιρετικό, ισχύει η προεπιλογή του provider. Αν είναι υποχρεωτικό, το plan σταματά με σφάλμα, όπως και όταν λείπει. Έτσι μια μεταβλητή μπορεί να αφήσει ένα argument στην προεπιλογή του provider, όπως θα δεις στο λυμένο παράδειγμα.

Ο τύπος `any` λέει στο Terraform να δεχτεί οποιαδήποτε τιμή και να βρει τον τύπο από την ίδια την τιμή. Μέσα σε συλλογή, όπως στο `list(any)`, όλα τα στοιχεία πρέπει να καταλήξουν σε έναν κοινό τύπο, όπως στο `tolist`. Δες τη διαφορά με δύο μεταβλητές στον φάκελο `team`.

any.tf

```
variable "loose" {
  type    = any
  default = ["web", 8080, true]
}

variable "mixed" {
  type    = list(any)
  default = ["web", 8080, true]
}
```



TERRAFORM CONSOLE

```
> type(var.loose)
tuple([
    string,
    number,
    bool,
])
> type(var.mixed)
list(string)
> var.mixed
tolist([
  "web",
  "8080",
  "true",
])
```

Με σκέτο `any` η τιμή κράτησε τον τύπο της, ένα tuple με τρεις διαφορετικούς τύπους. Με `list(any)` το Terraform έψαξε έναν τύπο για όλα τα στοιχεία, βρήκε το string και μετέτρεψε σε string το `8080` και το `true`. Το `any` είναι βολικό, αλλά χάνεις τον έλεγχο ενός ρητού τύπου.

optional σε attributes ενός object

Ένα object θέλει όλα τα attributes του. Για ρυθμίσεις με λογική προεπιλογή γράφεις `optional` μέσα στον ορισμό του object.

- `optional(number)` κάνει το attribute προαιρετικό. Αν λείπει, η τιμή του είναι `null`.
- `optional(number, 8080)` δίνει και προεπιλογή. Αν λείπει ή είναι `null`, η τιμή του είναι `8080`.

Το `optional` υπάρχει από το Terraform 1.3 και ανήκει στην ύλη της εξέτασης.

Λυμένο παράδειγμα, ένα αρχείο από ένα object

Θα φτιάξεις ένα αρχείο με όλες τις ρυθμίσεις του σε μία μεταβλητή τύπου object. Γύρνα στον φάκελο του κεφαλαίου και φτιάξε τον φάκελο `site`.

TERMINAL

```
cd ..
mkdir site
cd site
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "site" {
  type = object({
    name  = string
    pages = optional(list(string), ["index.html"])
    owner = object({
      name  = string
      email = string
    })
    mode = optional(string)
  })
}

resource "local_file" "home" {
  filename        = "${var.site.name}/${var.site.pages[0]}"
  content         = "Υπεύθυνος ${var.site.owner.name}, ${var.site.owner.email}\n"
  file_permission = var.site.mode
}

output "path" {
  value = local_file.home.filename
}
```

Διάβασε τον τύπο γραμμή προς γραμμή. Το `name` είναι υποχρεωτικό string. Το `pages` είναι προαιρετική λίστα με προεπιλογή το `["index.html"]`. Το `owner` είναι object μέσα στο object, με δύο υποχρεωτικά strings. Το `mode` είναι προαιρετικό string χωρίς προεπιλογή, άρα `null` όταν λείπει.

Στο resource το `filename` ενώνει το όνομα με την πρώτη σελίδα. Το `content` διαβάζει δύο επίπεδα βαθιά, για παράδειγμα `var.site.owner.name`. Κάθε τελεία ή αγκύλη κατεβαίνει ένα επίπεδο. Το `file_permission` παίρνει το `mode` όπως είναι, ακόμη κι αν είναι `null`.

Στο `terraform.tfvars` δώσε μόνο όσα είναι υποχρεωτικά.

terraform.tfvars

```
site = {
  name = "blog"
  owner = {
    name  = "Άννα"
    email = "anna@example.com"
  }
}
```

Η configuration έχει provider. Πριν από το `terraform init` ούτε το console ανοίγει.

TERRAFORM CONSOLE

```
> var.site.name
╷
│ Error: Inconsistent dependency lock file
│
│ The following dependency selections recorded in the lock file are
│ inconsistent with the current configuration:
│   - provider registry.terraform.io/hashicorp/local: required by this configuration but no version is selected
│
│ To make the initial dependency selections that will initialize the
│ dependency lock file, run:
│   terraform init
```

Η configuration χρειάζεται τον `hashicorp/local`, αλλά δεν υπάρχει ακόμη επιλεγμένη έκδοση στο lock file και το μήνυμα ζητά `terraform init`. Οι φάκελοι `explore` και `team` δεν είχαν provider και δεν το χρειάστηκαν.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.home will be created
  + resource "local_file" "home" {
      + content              = <<-EOT
            Υπεύθυνος Άννα, anna@example.com
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "blog/index.html"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + path = "blog/index.html"
local_file.home: Creating...
local_file.home: Creation complete after 0s [id=6508b856164388bac70e2c8fd08aa9819a6f5748]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

path = "blog/index.html"
```

Το `filename` έγινε `"blog/index.html"`, από το `name` και από την προεπιλογή του `pages`. Το `file_permission` είναι `"0777"`, αν και δεν το γράψαμε πουθενά. Το `mode` ήταν `null`, άρα το argument μετρήθηκε σαν να λείπει και ο provider έβαλε την προεπιλογή του.

Το console σε αυτόν τον φάκελο βλέπει και τη μεταβλητή και το state.

TERRAFORM CONSOLE

```
> var.site.pages
tolist([
  "index.html",
])
> var.site.mode
tostring(null)
> local_file.home.file_permission
"0777"
```

Το `pages` είναι η λίστα της προεπιλογής. Το `mode` τυπώνεται ως `tostring(null)`, δηλαδή `null` με τύπο string. Η τρίτη τιμή ήρθε από το state, γιατί το resource υπάρχει πια.

> **ΠΑΓΙΔΑ**
> Ένα attribute που δεν υπάρχει στον τύπο του object χάνεται χωρίς σφάλμα και χωρίς προειδοποίηση. Αν γράψεις `mdoe` αντί για `mode`, το Terraform πετά το `mdoe` και το `mode` μένει `null`.

Δοκίμασέ το. Γράψε το `terraform.tfvars` με το λάθος όνομα.

terraform.tfvars

```
site = {
  name = "blog"
  owner = {
    name  = "Άννα"
    email = "anna@example.com"
  }
  mdoe = "0600"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.home: Refreshing state... [id=6508b856164388bac70e2c8fd08aa9819a6f5748]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Ήθελες δικαιώματα `0600` και το plan λέει `No changes`. Όταν ένα plan δεν δείχνει μια αλλαγή που περίμενες, έλεγξε πρώτα τα ονόματα των attributes.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch09`. Όπου η configuration έχει provider, τρέξε πρώτα `terraform init`.

ΑΣΚΗΣΗ 9.1 Από λίστα σε set

Άνοιξε το console σε άδειο φάκελο. Μετάτρεψε τη λίστα `["web", "db", "web", "cache"]` σε set και μέτρησε με `length` πόσα στοιχεία έχει το set.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> toset(["web", "db", "web", "cache"])
toset([
  "cache",
  "db",
  "web",
])
> length(toset(["web", "db", "web", "cache"]))
3
```

ΑΣΚΗΣΗ 9.2 Λίστα από το terraform.tfvars

Γράψε μεταβλητή `cities` τύπου `list(string)` και δώσε της στο `terraform.tfvars` τις πόλεις `Αθήνα`, `Θεσσαλονίκη` και `Πάτρα`. Γράψε output `first` με την πρώτη πόλη και output `total` με το πλήθος τους. Κάνε apply και δείξε τα outputs με `terraform output`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
first = "Αθήνα"
total = 3
```

ΑΣΚΗΣΗ 9.3 Τιμή που δεν μετατρέπεται

Γράψε μεταβλητή `ports` τύπου `list(number)` και ένα output που την τυπώνει. Τρέξε `terraform plan` με `-var 'ports=["80", "443"]'` και μετά με `-var 'ports=["80", "http"]'`. Γιατί το πρώτο περνά και το δεύτερο όχι;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
│ Unsuitable value for var.ports set using -var="ports=...": a number is
│ required.
…
```

ΑΣΚΗΣΗ 9.4 Προεπιλογές με optional

Γράψε μεταβλητή `server` τύπου object με τα attributes `name` (string, υποχρεωτικό), `port` (number, προεπιλογή `8080`), `public` (bool, προεπιλογή `false`) και `owner` (string, προαιρετικό χωρίς προεπιλογή). Στο `terraform.tfvars` δώσε μόνο `name = "api"` και `public = true`. Δείξε στο console ολόκληρη τη μεταβλητή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> var.server
{
  "name" = "api"
  "owner" = tostring(null)
  "port" = 8080
  "public" = true
}
```

ΑΣΚΗΣΗ 9.5 Κλειδί από μεταβλητή

Γράψε μεταβλητή `sizes` τύπου `map(string)` με προεπιλογή `{ small = "t3.micro", large = "t3.large" }` και μεταβλητή `size` τύπου `string` με προεπιλογή `"small"`. Γράψε output `instance_type` που διαβάζει από το map την τιμή του κλειδιού που λέει το `size`. Κάνε apply με `-var size=large` και δείξε το output. Δοκίμασε μετά `terraform plan` με `-var size=medium`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
"t3.large"
```

ΑΣΚΗΣΗ 9.6 Αρχείο από object

Γράψε configuration με τον provider `local` και μία μεταβλητή `note` τύπου object με τα attributes `dir` (string), `files` (λίστα από strings), `text` (string) και `mode` (string, προεπιλογή `"0640"`). Ένα resource `local_file` γράφει το `text` στο αρχείο με όνομα το δεύτερο στοιχείο του `files`, μέσα στον φάκελο `dir`, με δικαιώματα `mode`. Στο `terraform.tfvars` δώσε `dir = "notes"`, `files = ["draft.txt", "final.txt"]` και ένα κείμενο. Τρέξε `terraform plan`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
      + file_permission      = "0640"
      + filename             = "notes/final.txt"
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 9.1

Μια μεταβλητή κρατά ονόματα περιοχών. Δεν πρέπει να υπάρχουν διπλότυπα και η σειρά δεν έχει σημασία. Ποιος τύπος ταιριάζει καλύτερα;

a. `list(string)`
b. `set(string)`
c. `tuple([string])`
d. `map(string)`

ΕΡΩΤΗΣΗ 9.2

Μια μεταβλητή περιγράφει έναν server με όνομα, αριθμό CPU και μια ένδειξη αν είναι δημόσιος. Ποιος τύπος κρατά κάθε τιμή με τον δικό της τύπο;

a. `map(string)`
b. `list(any)`
c. `object({ name = string, cpus = number, public = bool })`
d. `map(any)`

ΕΡΩΤΗΣΗ 9.3

Ποιοι από τους παρακάτω είναι δομικοί τύποι (structural types), δηλαδή επιτρέπουν διαφορετικό τύπο σε κάθε στοιχείο; Επίλεξε δύο.

a. `list`
b. `tuple`
c. `map`
d. `object`
e. `set`

ΕΡΩΤΗΣΗ 9.4

Ένα resource έχει `file_permission = var.mode` και η μεταβλητή `mode` έχει τιμή `null`. Τι κάνει το Terraform με το `file_permission`;

a. Το αντιμετωπίζει σαν να μη γράφτηκε, οπότε ισχύει η προεπιλογή του provider.
b. Στέλνει στον provider το string `"null"`.
c. Σταματά με σφάλμα τύπου.
d. Στέλνει στον provider κενό string.

ΕΡΩΤΗΣΗ 9.5

Μια μεταβλητή `app` έχει τύπο `object({ name = string, port = optional(number, 8080) })`. Το `terraform.tfvars` δίνει μόνο `name = "api"`. Ποια είναι η τιμή του `var.app.port`;

a. `null`
b. `0`
c. Δεν υπάρχει τιμή, γιατί το Terraform σταματά με σφάλμα για το attribute που λείπει.
d. `8080`

ΕΡΩΤΗΣΗ 9.6

Σωστό ή λάθος; Το `terraform console` μπορεί να αλλάξει ένα resource, αν γράψεις μια έκφραση που αναφέρεται σε αυτό.

a. Σωστό
b. Λάθος



ΚΕΦΑΛΑΙΟ 10

Εκφράσεις και συναρτήσεις

Μέχρι τώρα οι τιμές της configuration ήταν σταθερές ή αναφορές σε μεταβλητές και locals. Σε αυτό το κεφάλαιο μαθαίνεις να τις υπολογίζεις. Τελεστές, συνθήκες, templates, ενσωματωμένες συναρτήσεις και εκφράσεις `for` κάνουν τη configuration να προσαρμόζεται στα δεδομένα της χωρίς να αντιγράφεις κώδικα.

Τι είναι μια έκφραση

Ό,τι γράφεις δεξιά από το `=` ενός argument είναι **έκφραση** (expression). Το `"hello.txt"` είναι η πιο απλή, μια σταθερή τιμή. Το `var.environment` και το `local.names` είναι αναφορές, που γνώρισες στα κεφάλαια 7 και 8. Εδώ προστίθενται οι τελεστές, η έκφραση υπό όρους, τα string templates, οι κλήσεις συναρτήσεων, οι εκφράσεις `for` και τα splat. Κάθε έκφραση δίνει μια τιμή, οπότε μπαίνει μέσα σε μια μεγαλύτερη έκφραση όπου χωρά τιμή.

Το εργαλείο για να τις μάθεις είναι το `terraform console` του κεφαλαίου 9. Γράφεις μια έκφραση και βλέπεις αμέσως την τιμή της, χωρίς plan και χωρίς resources. Φτιάξε τον φάκελο `expr` με λίγα δεδομένα για να έχει το console κάτι να διαβάσει.

TERMINAL

```
mkdir expr
cd expr
```

main.tf

```
variable "environment" {
  type    = string
  default = "dev"
```

```
}

variable "users" {
  type = list(object({
    name = string
    team = string
    port = number
  }))
  default = [
    { name = "anna", team = "web", port = 8080 },
    { name = "nikos", team = "data", port = 5432 },
    { name = "eleni", team = "web", port = 8081 },
  ]
}

locals {
  names = ["anna", "nikos", "eleni"]
  ports = {
    web   = 8080
    db    = 5432
    cache = 6379
  }
}
```

Η configuration δεν χρειάζεται provider, οπότε το console ανοίγει χωρίς `terraform init`. Διαβάζει τις μεταβλητές με τα defaults τους και τα locals.

TERRAFORM CONSOLE

```
> var.environment
"dev"
> local.ports.web
8080
> var.users[1].name
"nikos"
```

Τελεστές

Οι αριθμητικοί τελεστές είναι `+`, `-`, `*`, `/` και `%`, το υπόλοιπο της διαίρεσης. Ο πολλαπλασιασμός και η διαίρεση γίνονται πριν από την πρόσθεση, όπως στα μαθηματικά. Οι παρενθέσεις αλλάζουν τη σειρά.

TERRAFORM CONSOLE

```
> 5 + 3 * 2
11
> (5 + 3) * 2
16
> 7 / 2
3.5
> 7 % 3
1
```

Το Terraform έχει έναν τύπο `number` για ακέραιους και δεκαδικούς. Γι' αυτό το `7 / 2` δίνει `3.5` και όχι `3`. Αν χρειάζεσαι ακέραιο, στρογγυλεύεις με τις συναρτήσεις `floor` ή `ceil` που θα δεις παρακάτω.

Οι τελεστές σύγκρισης `==`, `!=`, `<`, `>`, `<=` και `>=` δίνουν `true` ή `false`. Οι λογικοί τελεστές `&&` (και), `||` (ή) και `!` (όχι) συνδυάζουν τιμές bool.

TERRAFORM CONSOLE

```
> var.environment == "dev"
true
> local.ports.web > 8000 && local.ports.db > 8000
false
> !(var.environment == "prod")
true
> "5" == 5
false
> "5" + 1
6
```

Η δεύτερη γραμμή δίνει `false`, γιατί η θύρα `5432` δεν περνά το όριο και το `&&` θέλει και τις δύο πλευρές `true`. Οι δύο τελευταίες γραμμές μοιάζουν αντιφατικές. Το `"5" == 5` είναι `false`, ενώ το `"5" + 1` δίνει `6`.

> **ΠΑΓΙΔΑ**
> Το `==` συγκρίνει και τον τύπο. Ένα string δεν είναι ποτέ ίσο με αριθμό, ακόμη κι αν γράφουν το ίδιο. Ο τελεστής `+` μετατρέπει το `"5"` σε αριθμό επειδή δουλεύει μόνο με αριθμούς. Το `==` δεν μετατρέπει τίποτα. Το ίδιο ισχύει για list και tuple, οπότε το `tolist(["a"]) == ["a"]` είναι `false`. Όταν συγκρίνεις τιμές από διαφορετικές πηγές, μετάτρεψε πρώτα και τις δύο στον ίδιο τύπο.

Η έκφραση υπό όρους

Η **έκφραση υπό όρους** (conditional expression) γράφεται `συνθήκη ? α : β`. Αν η συνθήκη είναι `true`, η έκφραση παίρνει την τιμή `α`, αλλιώς την τιμή `β`. Είναι το if του Terraform μέσα σε μια τιμή.

TERRAFORM CONSOLE

```
> var.environment == "prod" ? 3 : 1
1
> var.environment == "prod" ? "m5.large" : "t3.micro"
"t3.micro"
> var.environment == "prod" ? "0600" : null
tostring(null)
```

Η τρίτη γραμμή είναι συνηθισμένο μοτίβο. Όπως είδες στο κεφάλαιο 9, ένα argument με τιμή `null` είναι σαν να μην γράφτηκε. Έτσι ένα argument ορίζεται μόνο σε production και στα άλλα περιβάλλοντα κρατά την προεπιλογή του provider. Το console γράφει `tostring(null)`, δηλαδή `null` με τύπο string. Ο λόγος φαίνεται αμέσως παρακάτω.

Οι δύο κλάδοι πρέπει να έχουν **συμβατό τύπο**. Η έκφραση έχει έναν τύπο που δεν εξαρτάται από τη συνθήκη, γιατί το Terraform πρέπει να τον ξέρει πριν μάθει την τιμή της συνθήκης. Αν οι κλάδοι διαφέρουν, προσπαθεί να τους μετατρέψει σε κοινό τύπο. Αν δεν βρει κοινό τύπο, σταματά με σφάλμα.

TERRAFORM CONSOLE

```
> true ? 1 : "δύο"
"1"
> true ? ["a"] : { a = 1 }
╷
│ Error: Inconsistent conditional result types
│
│   on <console-input> line 1:
│   (source code not available)
│
│ The true and false result expressions must have consistent types. The
│ 'true' value is tuple, but the 'false' value is object.
╵
```

Στην πρώτη γραμμή ο αριθμός και το string έχουν κοινό τύπο, το string. Το αποτέλεσμα είναι `"1"` με εισαγωγικά, δηλαδή string και όχι αριθμός. Για τον ίδιο λόγο το `null` πιο πάνω πήρε τον τύπο string του άλλου κλάδου. Στη δεύτερη γραμμή ένα tuple και ένα object δεν έχουν κοινό τύπο. Το μήνυμα `Inconsistent conditional result types` λέει ποιον τύπο βρήκε σε κάθε κλάδο.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Στο `συνθήκη ? α : β` οι δύο κλάδοι καταλήγουν πάντα σε έναν τύπο. Αν διαφέρουν, το Terraform μετατρέπει όπου μπορεί, αλλιώς βγάζει σφάλμα.

String templates

Κάθε string σε διπλά εισαγωγικά είναι **template**. Στο κεφάλαιο 2 είδες το interpolation. Το `${ }` υπολογίζει μια έκφραση και βάζει το αποτέλεσμα στη θέση της.

TERRAFORM CONSOLE

```
> "Περιβάλλον ${var.environment}, θύρα ${local.ports.web + 1}"
"Περιβάλλον dev, θύρα 8081"
```

Εκτός από τιμές, ένα template έχει και **directives** με `%{ }`. Το `%{ if συνθήκη }` μαζί με τα `%{ else }` και `%{ endif }` βάζει κείμενο υπό όρους. Το `%{ for x in συλλογή }` μαζί με το `%{ endfor }` επαναλαμβάνει κείμενο για κάθε στοιχείο.



TERRAFORM CONSOLE

```
> "Είσαι σε %{ if var.environment == "prod" }production%{ else }δοκιμές%{ endif }"
"Είσαι σε δοκιμές"
> "%{ for n in local.names }[${n}]%{ endfor }"
"[anna][nikos][eleni]"
> "%{ for i, n in local.names }${i}=${n} %{ endfor }"
"0=anna 1=nikos 2=eleni "
```

Το `for` με δύο ονόματα δίνει πρώτα τη θέση και μετά την τιμή. Σε map δίνει κλειδί και τιμή. Για να γράψεις τα σύμβολα `${` ή `%{` αυτούσια, διπλασιάζεις το πρώτο, `$${` και `%%{`, όπως στο κεφάλαιο 2. Χρειάζεται όταν το κείμενο είναι, για παράδειγμα, ένα shell script με δικές του μεταβλητές.

Heredoc και strip markers

Τα directives χρησιμεύουν κυρίως σε κείμενο πολλών γραμμών. Ένα heredoc, που γνώρισες στο κεφάλαιο 2, είναι κι αυτό template. Πρόσθεσε στον φάκελο ένα δεύτερο αρχείο με δύο εκδοχές της ίδιας λίστας.

templates.tf

```
locals {
  plain = <<EOT
Χρήστες
%{for n in local.names}
- ${n}
%{endfor}
EOT

  stripped = <<EOT
Χρήστες
%{for n in local.names~}
- ${n}
%{endfor~}
EOT
}
```

TERRAFORM CONSOLE

```
> local.plain
<<EOT
Χρήστες

- anna

- nikos

- eleni


EOT
> local.stripped
<<EOT
Χρήστες
- anna
- nikos
- eleni

EOT
```

Το console δείχνει ένα string πολλών γραμμών ως heredoc. Η κενή γραμμή πριν από το `EOT` σημαίνει ότι το κείμενο τελειώνει με αλλαγή γραμμής. Η πρώτη εκδοχή έχει επιπλέον κενή γραμμή πριν από κάθε χρήστη και μία ακόμη στο τέλος. Κάθε directive πιάνει μια δική του γραμμή. Η αλλαγή γραμμής μετά το `}` του είναι απλό κείμενο και μένει στο αποτέλεσμα. Η δεύτερη εκδοχή έχει **strip marker**, το `~` πριν από το `}`. Σβήνει τα κενά και την αλλαγή γραμμής που ακολουθούν το directive. Ένα `~` αμέσως μετά το `%{` σβήνει αντίστοιχα όσα προηγούνται.

ΣΗΜΕΙΩΣΗ

Στα αρχεία `.tf` το `terraform fmt` σβήνει τα κενά μέσα στα `${ }` και `%{ }`. Γι' αυτό το `templates.tf` γράφει `%{for n in local.names~}` και `${n}`. Στο console και στα αρχεία `.tftpl`, που το `fmt` δεν τα αγγίζει, θα τα βλέπεις συχνά με κενά, όπως `%{ for n in names ~}`. Για το Terraform είναι το ίδιο.

ΠΑΓΙΔΑ

Απόφυγε τα strip markers μέσα σε `<<-EOT`, το heredoc που αφαιρεί την εσοχή. Η αφαίρεση της εσοχής και τα strip markers δουλεύουν πάνω στα ίδια κενά και οι γραμμές βγαίνουν με εσοχή που δεν περιμένεις. Για templates με directives γράψε `<<EOT` χωρίς εσοχή ή, καλύτερα, ξεχωριστό αρχείο με την `templatefile` που θα δεις στο λυμένο παράδειγμα.

Συναρτήσεις

Μια συνάρτηση (function) καλείται με το όνομά της και τα arguments σε παρενθέσεις, `upper("abc")`. Τα arguments είναι κι αυτά εκφράσεις, οπότε οι κλήσεις μπαίνουν η μία μέσα στην άλλη. Η πιο εσωτερική υπολογίζεται πρώτη.

Το Terraform έχει πάνω από εκατό ενσωματωμένες συναρτήσεις (built-in functions). Δεν χρειάζεται να τις αποστηθίσεις. Χρειάζεται να ξέρεις τις κατηγορίες και τις πιο συχνές από την καθεμία, ώστε να καταλαβαίνεις μια έκφραση όταν τη διαβάζεις.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Στο Terraform δεν γράφεις δικές σου συναρτήσεις. Η γλώσσα δεν έχει block για ορισμό συνάρτησης. Υπάρχουν μόνο οι ενσωματωμένες και, από την έκδοση 1.8, οι συναρτήσεις που δίνουν οι providers, με τη σύνταξη `provider::όνομα::συνάρτηση`. Η εξέταση ρωτά συχνά αν υποστηρίζονται user-defined functions. Η απάντηση είναι όχι.

Strings

TERRAFORM CONSOLE

```
> upper("athens")
"ATHENS"
> lower("Prod")
"prod"
> title("hello world")
"Hello World"
> format("%s-%03d", "web", 7)
"web-007"
> join(", ", local.names)
"anna, nikos, eleni"
> split(",", "a,b,c")
tolist([
  "a",
  "b",
  "c",
])
> replace("v1.2.3", ".", "-")
"v1-2-3"
> trimspace("  κενά γύρω  ")
"κενά γύρω"
> substr("terraform", 0, 5)
"terra"
> startswith("prod-web", "prod")
true
```

Η `format` δουλεύει όπως η `printf` άλλων γλωσσών. Το `%s` δέχεται string, το `%d` ακέραιο και το `%03d` γεμίζει με μηδενικά ως τα τρία ψηφία. Η `join` ενώνει μια λίστα με διαχωριστικό και η `split` κάνει το αντίστροφο. Το `tolist(...)` γύρω από το αποτέλεσμα της `split` είναι ο τρόπος του console να πει ότι η τιμή είναι list και όχι tuple. Η `replace` αλλάζει κάθε εμφάνιση του κειμένου. Αν το δεύτερο argument γραφτεί μέσα σε κάθετες, όπως `"/[0-9]+/"`, είναι regular expression. Η `substr` παίρνει θέση αρχής και μήκος.

Αριθμοί

TERRAFORM CONSOLE

```
> min(5, 2, 9)
2
> max(local.ports.web, local.ports.db)
8080
> ceil(7 / 2)
4
> floor(7 / 2)
3
> abs(-7)
7
> max([5, 2, 9])
```

```
│ Error: Invalid function argument
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Invalid value for "numbers" parameter: number required, but have tuple.
╵
> max([5, 2, 9]...)
9
```

Οι `min` και `max` δέχονται αριθμούς ως χωριστά arguments. Μια λίστα ολόκληρη δεν τη δέχονται, γι' αυτό η έκτη γραμμή βγάζει σφάλμα με `number required, but have tuple`. Οι τρεις τελείες μετά από μια λίστα, `...`, την απλώνουν σε χωριστά arguments. Η τελευταία γραμμή είναι ο σωστός τρόπος.

Συλλογές

TERRAFORM CONSOLE

```
> length(local.names)
3
> concat(local.names, ["maria"])
[
  "anna",
  "nikos",
  "eleni",
  "maria",
]
> contains(local.names, "nikos")
true
> distinct(["a", "b", "a"])
tolist([
  "a",
  "b",
])
> flatten([["a", "b"], [], ["c"]])
[
  "a",
  "b",
  "c",
]
> slice(local.names, 1, 3)
[
  "nikos",
  "eleni",
]
> element(local.names, 3)
"anna"
> range(3)
tolist([
  0,
  1,
  2,
])
> sort(["b", "c", "a"])
tolist([
  "a",
  "b",
  "c",
])
```

```
> sort([10, 9, 100])
tolist([
  "10",
  "100",
  "9",
])
```

Η `length` μετρά στοιχεία λίστας, map ή χαρακτήρες string. Η `concat` ενώνει λίστες και η `flatten` ισιώνει λίστες μέσα σε λίστες. Η `contains` ελέγχει αν υπάρχει μια τιμή, η `distinct` πετά τα διπλά και η `range(3)` μετρά από το `0` έως το `2`. Η `slice` κρατά από τη θέση 1 έως πριν από τη θέση 3. Η `element` κάνει κύκλο. Η λίστα έχει τρία στοιχεία, οπότε η θέση 3 ξαναγυρίζει στο πρώτο, το `"anna"`. Η `sort` ταξινομεί πάντα ως strings. Γι' αυτό η τελευταία γραμμή έβαλε το `"100"` πριν από το `"9"` και επέστρεψε strings αντί για αριθμούς.

Για maps οι πιο χρήσιμες είναι οι παρακάτω.

TERRAFORM CONSOLE

```
> keys(local.ports)
[
  "cache",
  "db",
  "web",
]
> values(local.ports)
[
  6379,
  5432,
  8080,
]
> lookup(local.ports, "ftp", 21)
21
> merge(local.ports, { ftp = 21, web = 80 })
{
  "cache" = 6379
  "db" = 5432
  "ftp" = 21
  "web" = 80
}
> zipmap(["a", "b"], [1, 2])
{
  "a" = 1
  "b" = 2
}
```

Η `keys` επιστρέφει τα κλειδιά ταξινομημένα και η `values` τις τιμές με την ίδια σειρά των κλειδιών, γι' αυτό το `6379` του `cache` βγαίνει πρώτο. Η `lookup` δίνει την τιμή ενός κλειδιού ή την προεπιλογή του τρίτου argument αν το κλειδί λείπει. Η `merge` ενώνει maps. Όταν ένα κλειδί υπάρχει σε περισσότερα από ένα, κερδίζει το τελευταίο, οπότε το `web` έγινε `80`. Η `zipmap` φτιάχνει map από μια λίστα κλειδιών και μια λίστα τιμών.

Κωδικοποίηση

TERRAFORM CONSOLE

```
> jsonencode({ name = "web", ports = [80, 443] })
"{\"name\":\"web\",\"ports\":[80,443]}"
> jsondecode("{\"port\": 8080, \"debug\": true}")
{
  "debug" = true
  "port" = 8080
}
> yamlencode({ name = "web", ports = [80, 443] })
<<EOT
"name": "web"
"ports":
- 80
- 443

EOT
> base64encode("hello")
"aGVsbG8="
> base64decode("aGVsbG8=")
"hello"
```

Η `jsonencode` μετατρέπει μια τιμή του Terraform σε string JSON. Το console δείχνει τα εισαγωγικά του JSON ως `\"` επειδή όλο το αποτέλεσμα είναι ένα string. Η `jsondecode` κάνει το αντίστροφο και δίνει object, οπότε μετά γράφεις `.port` όπως σε κάθε object. Η `yamlencode` βγάζει κείμενο πολλών γραμμών και το console το δείχνει ως heredoc. Οι `base64encode` και `base64decode` χρειάζονται συχνά σε APIs που θέλουν δεδομένα σε base64.

Αρχεία

Γράψε ένα μικρό αρχείο κειμένου στον φάκελο.

greeting.txt

```
Καλημέρα από αρχείο
```

TERRAFORM CONSOLE

```
> file("greeting.txt")
<<EOT
Καλημέρα από αρχείο

EOT
> fileexists("greeting.txt")
true
> fileexists("missing.txt")
false
> file("missing.txt")
│ Error: Invalid function argument
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Invalid value for "path" parameter: no file exists at "missing.txt"; this
│ function works only with files that are distributed as part of the
│ configuration source code, so if this file will be created by a resource in
```

```
  this configuration you must instead obtain this result from an attribute of
  that resource.

> path.module
"."
> abspath(path.module)
"/home/you/tfbook/ch10/expr"
```

Η `file` διαβάζει όλο το αρχείο σε string και η `fileexists` ελέγχει αν υπάρχει. Το σφάλμα της τέταρτης γραμμής εξηγεί έναν σημαντικό περιορισμό. Οι συναρτήσεις αρχείων διαβάζουν αρχεία που υπάρχουν ήδη μαζί με τον κώδικα. Ένα αρχείο που θα φτιάξει κάποιο resource της ίδιας configuration δεν υπάρχει όταν υπολογίζονται οι εκφράσεις. Την τιμή του τη διαβάζεις από attribute του resource.

Το `path.module` είναι ο φάκελος του module όπου βρίσκεται η έκφραση. Στο root module είναι το `"."` και η `abspath` το κάνει απόλυτο path. Η διαφορά φαίνεται στα modules του κεφαλαίου 15, που ζουν σε υποφάκελους και πρέπει να βρίσκουν τα δικά τους αρχεία. Γράφε λοιπόν πάντα `"${path.module}/αρχείο"` αντί για σκέτο όνομα.

Ημερομηνίες

TERRAFORM CONSOLE

```
> timestamp()
"2026-10-08T02:27:24Z"
> formatdate("DD/MM/YYYY hh:mm", "2026-10-07T09:30:00Z")
"07/10/2026 09:30"
> timeadd("2026-10-07T09:30:00Z", "48h")
"2026-10-09T09:30:00Z"
```

Η `timestamp` δίνει την τρέχουσα ώρα σε UTC, στη μορφή RFC 3339. Η δική σου τιμή θα είναι διαφορετική. Η `formatdate` τη γράφει με όποια μορφή θέλεις. Το `DD` είναι η μέρα, το `MM` ο μήνας, το `YYYY` το έτος και τα `hh` και `mm` η ώρα και τα λεπτά. Η `timeadd` προσθέτει διάρκεια όπως `"48h"` ή `"-90m"`.

Η `timestamp` έχει μια παγίδα. Η τιμή της αλλάζει σε κάθε εκτέλεση, οπότε το Terraform δεν την υπολογίζει στο plan. Τη θεωρεί άγνωστη μέχρι το apply. Δες τι σημαίνει αυτό για ένα resource, σε δικό του φάκελο.

TERMINAL

```
mkdir ../stamp
cd ../stamp
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "stamp" {
  filename = "stamp.txt"
  content  = "Δημιουργήθηκε ${timestamp()}\n"
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

Το apply έφτιαξε το αρχείο. Τρέξε τώρα plan χωρίς να αλλάξεις τίποτα.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.stamp: Refreshing state... [id=bc877ea6b4f5ca4bc423beed384c573e0b9238cb]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.stamp must be replaced
-/+ resource "local_file" "stamp" {
      ~ content              = <<-EOT
            Δημιουργήθηκε 2026-10-08T02:27:27Z
        EOT -> (known after apply) # forces replacement
      …
      ~ id                   = "bc877ea6b4f5ca4bc423beed384c573e0b9238cb" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το παλιό `content` γράφει την ώρα του apply και η νέα τιμή είναι `(known after apply)`. Το Terraform δεν μπορεί να ξέρει ότι θα βγει ίδια, οπότε σημειώνει αλλαγή με `# forces replacement`. Η περίληψη λέει `1 to add` και `1 to destroy`. Αυτό θα γίνεται σε κάθε plan, για πάντα.

> **ΠΑΓΙΔΑ**
> Μη βάζεις `timestamp()` σε argument resource, για παράδειγμα σε όνομα ή σε ετικέτα. Κάθε plan θα δείχνει αλλαγή και δεν θα φτάσεις ποτέ στο `No changes`. Αν χρειάζεσαι την ώρα δημιουργίας, το κεφάλαιο 14 δείχνει πώς το `ignore_changes` αγνοεί τις επόμενες αλλαγές ενός argument.

TERMINAL

```
cd ../expr
```

Hash

TERRAFORM CONSOLE

```
> sha256("hello")
"2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
> md5("hello")
"5d41402abc4b2a76b9719d911017c592"
> sha1("Γεια σου, Terraform\n")
"385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07"
```

Ένα hash είναι αποτύπωμα του περιεχομένου. Η ίδια είσοδος δίνει πάντα το ίδιο αποτέλεσμα και η παραμικρή αλλαγή δίνει τελείως άλλο. Η τρίτη γραμμή δίνει `385b0374cb7d9372c1ddc72aa0ba9c9a75dc4d07`, το ίδιο `id` που πήρε το `local_file.hello` στο κεφάλαιο 1. Ο provider `local` χρησιμοποιεί το SHA1 του περιεχομένου ως `id`.

Μετατροπές τύπων, can και try

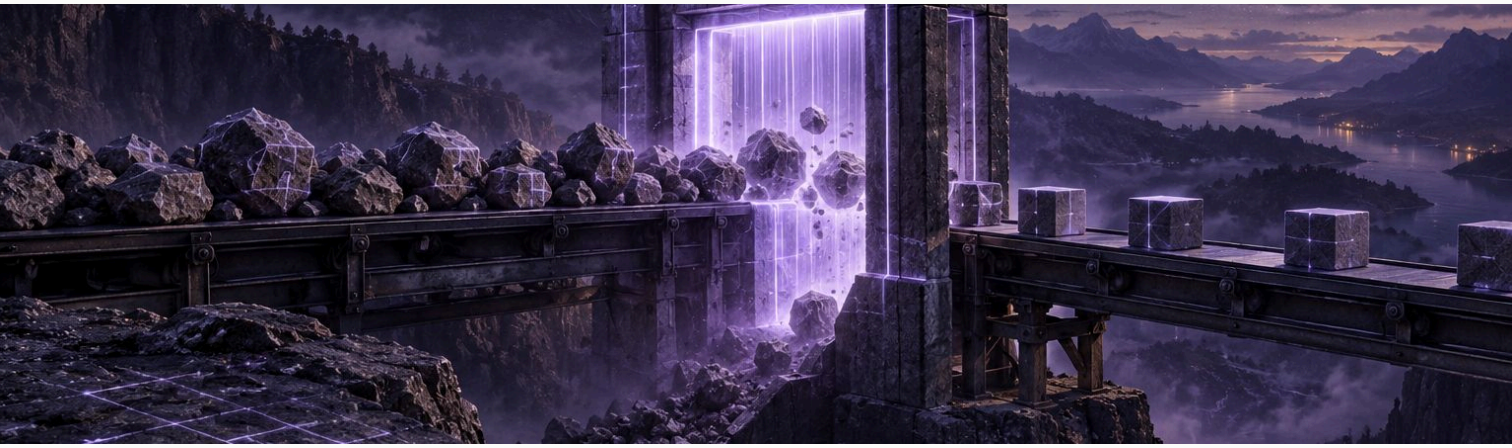
TERRAFORM CONSOLE

```
> tostring(42)
"42"
> tonumber("42")
42
> tolist(toset(["b", "a", "b"]))
tolist([
  "a",
  "b",
])
> can(tonumber("abc"))
false
> try(tonumber("abc"), 0)
0
> try(var.users[5].name, "κανείς")
"κανείς"
```

Οι συναρτήσεις `tostring`, `tonumber`, `tolist`, `toset` και `tomap` κάνουν ρητά τις μετατροπές που είδες στο κεφάλαιο 9. Η `toset` πέταξε το διπλό `"b"`.

Η `can` δέχεται μια έκφραση και λέει αν υπολογίζεται χωρίς σφάλμα. Το `"abc"` δεν γίνεται αριθμός, οπότε επιστρέφει `false`. Η `try` δέχεται πολλές εκφράσεις και επιστρέφει την πρώτη που δεν βγάζει σφάλμα. Η λίστα `var.users` έχει τρία στοιχεία, οπότε το `var.users[5]` αποτυγχάνει και η `try` δίνει την προεπιλογή. Ταιριάζει σε δεδομένα από JSON ή YAML, όπου ένα πεδίο μπορεί να λείπει.

Οι `can` και `try` πιάνουν μόνο σφάλματα που προκύπτουν από τις τιμές. Ένα λάθος στον ίδιο τον κώδικα, όπως αναφορά σε local που δεν έχει δηλωθεί, βγάζει σφάλμα ακόμη και μέσα σε `try`.



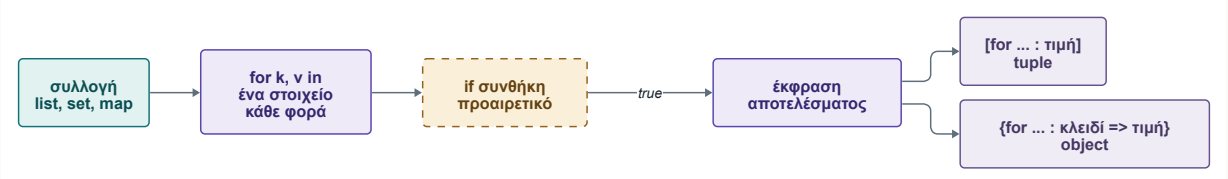
Εκφράσεις for

Μια **έκφραση for** διαβάζει μια συλλογή στοιχείο προς στοιχείο και φτιάχνει νέα συλλογή. Έχει δύο μορφές, που διαφέρουν στις αγκύλες.

ΕΞΟΔΟΣ

```
[for ΣΤΟΙΧΕΙΟ in ΣΥΛΛΟΓΗ : ΤΙΜΗ if ΣΥΝΘΗΚΗ]
{for ΣΤΟΙΧΕΙΟ in ΣΥΛΛΟΓΗ : ΚΛΕΙΔΙ => ΤΙΜΗ if ΣΥΝΘΗΚΗ}
```

Με τετράγωνες αγκύλες βγαίνει tuple, με μία τιμή ανά στοιχείο. Με άγκιστρα βγαίνει object και κάθε στοιχείο δίνει ένα ζεύγος `κλειδί => τιμή`. Το `if` είναι προαιρετικό και κρατά μόνο τα στοιχεία για τα οποία η συνθήκη είναι `true`.



Σχήμα 10.1 · Μια έκφραση for διαβάζει μια συλλογή, κρατά όσα στοιχεία περνούν το προαιρετικό if και φτιάχνει tuple με τετράγωνες αγκύλες ή object με άγκιστρα.

TERRAFORM CONSOLE

```
> [for n in local.names : upper(n)]
[
  "ANNA",
  "NIKOS",
  "ELENI",
]
> [for n in local.names : n if n != "nikos"]
[
  "anna",
  "eleni",
]
> {for n in local.names : n => length(n)}
{
```

```
  "anna" = 4
  "eleni" = 5
  "nikos" = 5
}
> {for k, v in local.ports : k => v if v > 6000}
{
  "cache" = 6379
  "web" = 8080
}
> [for k, v in local.ports : "${k}=${v}"]
[
  "cache=6379",
  "db=5432",
  "web=8080",
]
```

Οι δύο πρώτες γραμμές κρατούν τη σειρά της λίστας. Η τρίτη φτιάχνει object με κλειδί το όνομα και τιμή το μήκος του. Στις δύο τελευταίες η συλλογή είναι map, οπότε το `for` δίνει δύο ονόματα, κλειδί και τιμή. Η τέταρτη κρατά μόνο τις θύρες πάνω από `6000` και η πέμπτη μετατρέπει το map σε λίστα από strings. Τα κλειδιά ενός map ή object διαβάζονται και εμφανίζονται πάντα ταξινομημένα. Γι' αυτό στην τρίτη γραμμή το `eleni` βγαίνει πριν από το `nikos` και στην πέμπτη το `cache` βγαίνει πρώτο.

Τι γίνεται αν δύο στοιχεία δώσουν το ίδιο κλειδί; Η Άννα και η Ελένη είναι στην ομάδα `web`.

TERRAFORM CONSOLE

```
> {for u in var.users : u.team => u.name}

│ Error: Duplicate object key
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Two different items produced the key "web" in this 'for' expression. If
│ duplicates are expected, use the ellipsis (...) after the value expression
│ to enable grouping by key.

> {for u in var.users : u.team => u.name...}
{
  "data" = [
    "nikos",
  ]
  "web" = [
    "anna",
    "eleni",
  ]
}
```

Η πρώτη γραμμή σταματά με `Duplicate object key`, γιατί ένα object δεν έχει δύο φορές το ίδιο κλειδί. Οι τρεις τελείες μετά την τιμή ενεργοποιούν την ομαδοποίηση (grouping). Κάθε κλειδί παίρνει τότε μια λίστα με όλες τις τιμές που του αντιστοιχούν. Το μήνυμα του σφάλματος προτείνει μόνο του αυτή τη λύση.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Οι αγκύλες ορίζουν το αποτέλεσμα. Το `[for ...]` δίνει tuple με μία τιμή ανά στοιχείο. Το `{for ...}` δίνει object και θέλει `κλειδί => τιμή`. Το `if` στο τέλος φιλτράρει και το `...` μετά την τιμή ομαδοποιεί τα διπλά κλειδιά.

Splat

Η πιο συχνή έκφραση `for` παίρνει ένα attribute από κάθε object μιας λίστας. Γι' αυτή υπάρχει συντομογραφία, η **έκφραση splat** με `[*]`.

TERRAFORM CONSOLE

```
> var.users[*].name
tolist([
  "anna",
  "nikos",
  "eleni",
])
> [for u in var.users : u.name]
[
  "anna",
  "nikos",
  "eleni",
]
> max(var.users[*].port...)
8081
```

Οι δύο πρώτες γραμμές δίνουν τα ίδια ονόματα. Η τρίτη συνδυάζει splat με τις τρεις τελείες για να βρει τη μεγαλύτερη θύρα. Το splat δουλεύει σε list, set και tuple. Σε map δεν κάνει αυτό που περιμένεις, οπότε εκεί γράφεις `for`. Στο κεφάλαιο 11 θα το χρησιμοποιήσεις σε πολλά resources μαζί. Σε παλιότερο κώδικα θα δεις και τη μορφή `var.users.*.name`, που δουλεύει ακόμη.

Λυμένο παράδειγμα με templatefile και jsonencode

Ένα πραγματικό project χρειάζεται συχνά αρχεία ρυθμίσεων που αλλάζουν ανά περιβάλλον. Εδώ φτιάχνεις δύο. Ένα μήνυμα καλωσορίσματος από template και ένα αρχείο JSON για μια εφαρμογή.

TERMINAL

```
cd ..
mkdir site
cd site
```

Το template μπαίνει σε δικό του αρχείο με κατάληξη `.tftpl`. Η σύνταξη είναι ίδια με τα templates που είδες, χωρίς εισαγωγικά γύρω γύρω.

motd.tftpl

```
Καλώς ήρθες στο ${upper(app)} (${environment})
%{ for name, port in services ~}
* ${name} στη θύρα ${port}
%{ endfor ~}
%{ if environment == "prod" ~}
Προσοχή, είσαι σε production.
%{ else ~}
Περιβάλλον δοκιμών.
%{ endif ~}
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "environment" {
  type    = string
  default = "dev"
}

locals {
  app      = "shop"
  replicas = var.environment == "prod" ? 3 : 1
  services = {
    web = 8080
    api = 9090
  }
}

resource "local_file" "motd" {
  filename = "${path.module}/out/motd.txt"
  content = templatefile("${path.module}/motd.tftpl", {
    app         = local.app
    environment = var.environment
    services    = local.services
  })
}

resource "local_file" "config" {
  filename = "${path.module}/out/config.json"
  content = jsonencode({
    app      = local.app
    replicas = local.replicas
    debug    = var.environment != "prod"
    ports    = values(local.services)
  })
}
```

Διάβασε τον κώδικα από πάνω προς τα κάτω.

- Το local `replicas` είναι έκφραση υπό όρους. Τρία αντίγραφα σε production, ένα αλλού.
- Το `filename` χρησιμοποιεί interpolation με το `path.module`, ώστε το path να μετράει από τον φάκελο του module.
- Η `templatefile` παίρνει το path του template και ένα object με μεταβλητές. Κάθε κλειδί του object γίνεται μεταβλητή μέσα στο template. Γι’ αυτό το template γράφει `${app}` και όχι `${local.app}`. Βλέπει μόνο όσα του δίνεις.
- Στο template το `%{ for name, port in services ~}` γράφει μία γραμμή ανά service. Το `%{ if }` διαλέγει την τελευταία γραμμή. Τα strip markers σβήνουν τις αλλαγές γραμμής των directives, ώστε να μη μείνουν κενές γραμμές.
- Η `jsonencode` μετατρέπει ένα object του Terraform σε κείμενο JSON. Το `debug` είναι σύγκριση, οπότε γίνεται bool του JSON. Το `ports` είναι οι τιμές του map `services`.

Κάνε init και δες το plan. Τα checksums και τα permissions κόβονται για να φαίνονται τα περιεχόμενα.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.config will be created
  + resource "local_file" "config" {
      + content              = jsonencode(
            {
              + app      = "shop"
              + debug    = true
              + ports    = [
                  + 9090,
                  + 8080,
                ]
              + replicas = 1
            }
        )
      …
      + filename             = "./out/config.json"
      + id                   = (known after apply)
    }

  # local_file.motd will be created
  + resource "local_file" "motd" {
      + content              = <<-EOT
            Καλώς ήρθες στο SHOP (dev)
            * api στη θύρα 9090
            * web στη θύρα 8080
            Περιβάλλον δοκιμών.
        EOT
      …
      + filename             = "./out/motd.txt"
      + id                   = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το plan δείχνει το αποτέλεσμα των εκφράσεων πριν φτιαχτεί οτιδήποτε. Το `filename` έγινε `./out/config.json`, γιατί το `path.module` είναι `.`. Το template έχει ήδη αποδοθεί και το `content` του `local_file.motd` γράφει το τελικό κείμενο. Για το `local_file.config` το Terraform αναγνωρίζει ότι το string είναι JSON και το δείχνει ως `jsonencode(...)` με τα πεδία του ένα ένα. Το `api` βγαίνει πριν από το `web` και στα δύο

αρχεία. Τόσο το `for` του template όσο και η `values` διαβάζουν το map με ταξινομημένα κλειδιά. Γι' αυτό το `ports` είναι `9090` και μετά `8080`.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
cat out/motd.txt
cat out/config.json
```

ΕΞΟΔΟΣ

```
Καλώς ήρθες στο SHOP (dev)
* api στη θύρα 9090
* web στη θύρα 8080
Περιβάλλον δοκιμών.
{"app":"shop","debug":true,"ports":[9090,8080],"replicas":1}
```

Το JSON βγαίνει σε μία γραμμή, χωρίς κενά και με ταξινομημένα κλειδιά. Αυτή είναι η μορφή της `jsonencode`. Για μια εφαρμογή που διαβάζει JSON είναι ό,τι χρειάζεται. Τώρα άλλαξε περιβάλλον με `-var` από το κεφάλαιο 7.

TERMINAL

```
terraform apply -auto-approve -var environment=prod
```

TERMINAL

```
cat out/motd.txt
cat out/config.json
```

ΕΞΟΔΟΣ

```
Καλώς ήρθες στο SHOP (prod)
* api στη θύρα 9090
* web στη θύρα 8080
Προσοχή, είσαι σε production.
{"app":"shop","debug":false,"ports":[9090,8080],"replicas":3}
```

Μία μεταβλητή άλλαξε και τα δύο αρχεία. Το template έγραψε `(prod)` και διάλεξε τον άλλο κλάδο του `%{ if }`. Στο JSON το `replicas` έγινε `3` και το `debug` έγινε `false`.

Συναρτήσεις από providers

Από την έκδοση 1.8 ένας provider μπορεί να δίνει δικές του συναρτήσεις. Ο `hashicorp/local` 2.9.1 δίνει τη `direxists`, που ελέγχει αν υπάρχει φάκελος. Ο φάκελος `site` έχει ήδη τον provider και έχει γίνει `init`, οπότε το console τη βρίσκει.

TERRAFORM CONSOLE

```
> provider::local::direxists("out")
true
> provider::local::direxists("missing")
false
```

Η κλήση γράφεται `provider::`, το τοπικό όνομα του provider από το `required_providers`, `::` και το όνομα της συνάρτησης. Ο provider πρέπει να είναι δηλωμένος στο `required_providers` του module. Δεν αρκεί να τον χρησιμοποιεί κάποιο resource. Πρέπει επίσης να είναι εγκατεστημένος. Χωρίς `terraform init` το console δεν ανοίγει καν. Τις συναρτήσεις αυτές τις γράφουν οι δημιουργοί του provider, όπως και τα resources του. Εσύ μέσα στην configuration εξακολουθείς να μην ορίζεις συναρτήσεις.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch10`. Όσες ζητούν μόνο εκφράσεις λύνονται στο `terraform console` και δεν χρειάζονται `init`, εκτός αν η configuration τους έχει provider.

ΑΣΚΗΣΗ 10.1 Ζυγός ή μονός

Γράψε στο console μια έκφραση που επιστρέφει `"ζυγός"` ή `"μονός"` για έναν αριθμό. Δοκίμασέ τη με το `7` και με το `10`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> 7 % 2 == 0 ? "ζυγός" : "μονός"
"μονός"
> 10 % 2 == 0 ? "ζυγός" : "μονός"
"ζυγός"
```

ΑΣΚΗΣΗ 10.2 Όνομα χωρίς κενά

Από το string `"  Web Server 01  "` φτιάξε με μία έκφραση το `"web-server-01"`. Χρησιμοποίησε τρεις συναρτήσεις string.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> replace(lower(trimspace("  Web Server 01  ")), " ", "-")
"web-server-01"
```

ΑΣΚΗΣΗ 10.3 Φίλτρο σε map

Ξεκίνα με το παρακάτω `main.tf`. Στο console γράψε μια έκφραση `for` που επιστρέφει λίστα με τα ονόματα, σε κεφαλαία, των services με θύρα πάνω από `1024`.

main.tf

```
variable "services" {
  type = map(number)
  default = {
    web   = 80
    api   = 8080
    admin = 9000
    db    = 5432
  }
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> [for name, port in var.services : upper(name) if port > 1024]
[
  "ADMIN",
  "API",
  "DB",
]
```

ΑΣΚΗΣΗ 10.4 Ομάδες

Ξεκίνα με το παρακάτω `main.tf`. Στο console βρες πρώτα όλα τα emails με splat. Μετά φτιάξε object που έχει κλειδί την ομάδα και τιμή τη λίστα με τα ονόματα των μελών της.

main.tf

```
variable "users" {
  type = list(object({
    name  = string
    team  = string
    email = string
  }))
  default = [
    { name = "anna", team = "web", email = "anna@example.com" },
    { name = "nikos", team = "data", email = "nikos@example.com" },
    { name = "eleni", team = "web", email = "eleni@example.com" },
    { name = "maria", team = "data", email = "maria@example.com" },
  ]
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> var.users[*].email
tolist([
  "anna@example.com",
  "nikos@example.com",
  "eleni@example.com",
  "maria@example.com",
])
> {for u in var.users : u.team => u.name...}
{
  "data" = [
    "nikos",
    "maria",
  ]
  "web" = [
    "anna",
    "eleni",
  ]
}
```

ΑΣΚΗΣΗ 10.5 Ρυθμίσεις από JSON

Γράψε στον φάκελο της άσκησης το παρακάτω `settings.json`. Στο console διάβασε τον αριθμό `replicas`, μέτρησε τα `features` και βρες το `region`. Αν το `region` λείπει από το αρχείο, η έκφραση πρέπει να δίνει `"eu-central-1"`.

settings.json

```
{
  "app": "shop",
  "replicas": 2,
  "features": ["cart", "search", "reviews"]
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
> jsondecode(file("settings.json")).replicas
2
> length(jsondecode(file("settings.json")).features)
3
> try(jsondecode(file("settings.json")).region, "eu-central-1")
"eu-central-1"
```

ΑΣΚΗΣΗ 10.6 Αρχείο hosts από template

Γράψε ένα template `hosts.tftpl` και ένα `main.tf` με τον provider `local`. Η μεταβλητή `hosts` είναι map από όνομα σε διεύθυνση IP, με default `web = "10.0.1.10"`, `db = "10.0.2.20"` και `cache = "10.0.2.30"`. Ένα `local_file` γράφει το `hosts.txt` με μία γραμμή `IP όνομα` για κάθε host, χωρίς κενές γραμμές. Στο τέλος έρχεται η γραμμή `# σύνολο 3`, με αριθμό που υπολογίζεται. Κάνε apply και δείξε το αρχείο με `cat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
10.0.2.30 cache
10.0.2.20 db
10.0.1.10 web
# σύνολο 3
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 10.1

Μια ομάδα θέλει μια δική της συνάρτηση που φτιάχνει ονόματα resources με σταθερό μορφότυπο και να την καλεί από πολλά σημεία της configuration. Ποια πρόταση ισχύει;

a. Ορίζει τη συνάρτηση με ένα block `function` σε οποιοδήποτε αρχείο `.tf`.
b. Ορίζει τη συνάρτηση μέσα σε ένα block `locals` με παραμέτρους.
c. Το Terraform δεν υποστηρίζει συναρτήσεις του χρήστη. Υπάρχουν μόνο οι ενσωματωμένες και όσες δίνουν οι providers.
d. Γράφει τη συνάρτηση σε αρχείο `.tftpl` και την καλεί με το όνομά της.

ΕΡΩΤΗΣΗ 10.2

Η μεταβλητή `var.enabled` είναι τύπου `bool`. Τι συμβαίνει με την έκφραση `var.enabled ? ["a"] : { a = 1 }`;

a. Επιστρέφει tuple ή object ανάλογα με την τιμή της μεταβλητής.
b. Βγάζει σφάλμα, γιατί οι δύο κλάδοι δεν έχουν κοινό τύπο.
c. Επιστρέφει πάντα το tuple, γιατί είναι ο πρώτος κλάδος.
d. Μετατρέπει το object σε tuple και επιστρέφει tuple.

ΕΡΩΤΗΣΗ 10.3

Τι επιστρέφει η έκφραση `{for s in ["a", "b"] : s => upper(s)}`;

a. Το tuple `["A", "B"]`.
b. Μια λίστα με δύο objects, `[{ a = "A" }, { b = "B" }]`.
c. Σφάλμα, γιατί τα άγκιστρα θέλουν map ως είσοδο.
d. Ένα object με κλειδιά `a` και `b` και τιμές `"A"` και `"B"`.

ΕΡΩΤΗΣΗ 10.4

Η `var.servers` είναι λίστα από objects, καθένα με attribute `ip`. Ποια έκφραση δίνει το ίδιο αποτέλεσμα με το `var.servers[*].ip`;

a. `[for s in var.servers : s.ip]`
b. `{for s in var.servers : s => s.ip}`
c. `var.servers.ip`
d. `values(var.servers).ip`

ΕΡΩΤΗΣΗ 10.5

Ένα resource έχει `content = "Build ${timestamp()}"`. Έγινε apply και από τότε δεν άλλαξε τίποτα. Τι δείχνει το επόμενο `terraform plan`;

a. `No changes`, γιατί η configuration δεν άλλαξε.
b. Σφάλμα, γιατί η `timestamp` δεν επιτρέπεται σε argument resource.
c. Αλλαγή στο resource, γιατί η τιμή της `timestamp` είναι άγνωστη μέχρι το apply και αλλάζει σε κάθε εκτέλεση.
d. Αλλαγή μόνο αν έχει περάσει τουλάχιστον μία μέρα από το προηγούμενο apply.

ΕΡΩΤΗΣΗ 10.6

Ποιες εκφράσεις επιστρέφουν `7`; Επίλεξε δύο.

a. `max([3, 7, 5])`
b. `max([3, 7, 5]...)`
c. `max(3, 7, 5)`
d. `length([3, 7, 5])`
e. `lookup({ a = 7 }, "b", 0)`



ΚΕΦΑΛΑΙΟ 11

count, for_each και dynamic blocks

Ως τώρα κάθε block `resource` έφτιαχνε ένα αντικείμενο. Για τρία αρχεία έγραφες τρία blocks. Σε αυτό το κεφάλαιο ένα block φτιάχνει όσα αντικείμενα θέλεις με τα `count` και `for_each`, ένα resource εμφανίζεται μόνο όταν το ζητήσεις και τα επαναλαμβανόμενα nested blocks γράφονται μία φορά με το `dynamic`. Μαθαίνεις επίσης γιατί η επιλογή ανάμεσα σε `count` και `for_each` κρίνει αν μια μικρή αλλαγή σβήνει ένα αντικείμενο ή περισσότερα.

Meta-arguments

Μέσα σε ένα block `resource` τα περισσότερα arguments ανήκουν στον provider. Το `filename` και το `content` τα ορίζει ο `local`. Υπάρχουν όμως και arguments που τα διαβάζει το ίδιο το Terraform και δουλεύουν σε κάθε τύπο resource. Λέγονται **meta-arguments**. Γνώρισες ήδη το `depends_on` στο κεφάλαιο 6 και το `provider` στο κεφάλαιο 5. Το `lifecycle` έρχεται στο κεφάλαιο 14. Εδώ βλέπεις τα δύο που πολλαπλασιάζουν ένα block, το `count` και το `for_each`.

Ένα block με `count` ή `for_each` δεν είναι πια ένα αντικείμενο. Είναι μια ομάδα από **instances**, το καθένα με δική του διεύθυνση στο state. Η διεύθυνση παίρνει στο τέλος έναν αριθμό, όπως `local_file.page[0]`, ή ένα κλειδί, όπως `local_file.page["home"]`. Το κεφάλαιο 6 σου έδειξε αυτές τις μορφές. Εδώ βλέπεις από πού προέρχονται.

count

Το `count` παίρνει έναν ακέραιο αριθμό και φτιάχνει τόσα instances. Μέσα στο block το `count.index` είναι η θέση του instance που φτιάχνεται, από το `0` ως το `count` μείον ένα. Φτιάξε τον πρώτο φάκελο.

TERMINAL

```
mkdir pages
cd pages
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "pages" {
  type    = list(string)
  default = ["home", "about", "contact"]
}

resource "local_file" "page" {
  count    = length(var.pages)
  filename = "${path.module}/out/${var.pages[count.index]}.html"
  content  = "<h1>${var.pages[count.index]}</h1>\n"
}
```

Το `count` είναι το μήκος της λίστας, δηλαδή `3`. Για κάθε instance το `count.index` διαλέγει ένα στοιχείο με index, όπως έμαθες στο κεφάλαιο 9. Το instance `0` παίρνει το `"home"`, το `1` το `"about"` και το `2` το `"contact"`.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.page[0] will be created
…
      + filename             = "./out/home.html"
…
  # local_file.page[1] will be created
…
      + filename             = "./out/about.html"
…
  # local_file.page[2] will be created
…
      + filename             = "./out/contact.html"
…
Plan: 3 to add, 0 to change, 0 to destroy.
…
```

Ένα block έγινε τρία instances στο plan. Κάθε γραμμή με `#` γράφει τη δική του διεύθυνση με αριθμό σε αγκύλες. Η περίληψη λέει `3 to add`.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
terraform state list
ls out
```

ΕΞΟΔΟΣ

```
local_file.page[0]
local_file.page[1]
local_file.page[2]
about.html
contact.html
home.html
```

Το state έχει τρεις εγγραφές, μία για κάθε instance. Στον δίσκο υπάρχουν τρία αρχεία.

Αναφορές σε instances του count

Ένα resource με `count` είναι λίστα. Ένα instance το διαβάζεις με index και όλα μαζί με το splat `[*]` του κεφαλαίου 10. Το console διαβάζει και το state, οπότε βλέπει τα attributes των αρχείων που μόλις φτιάχτηκαν.

TERRAFORM CONSOLE

```
> local_file.page[0].filename
"./out/home.html"
> local_file.page[*].filename
[
  "./out/home.html",
  "./out/about.html",
  "./out/contact.html",
]
> length(local_file.page)
3
```

Το `local_file.page[0]` είναι ένα instance και έχει `filename`. Το `local_file.page` χωρίς index είναι όλη η λίστα. Το splat παίρνει το ίδιο attribute από κάθε στοιχείο της και η `length` μετρά τα instances.

ΣΗΜΕΙΩΣΗ

Το `count = 0` είναι επιτρεπτό και σημαίνει κανένα instance. Πάνω σε αυτό στηρίζεται η δημιουργία υπό όρους που θα δεις παρακάτω. Το `count` δεν δέχεται αρνητικό ή δεκαδικό αριθμό.

Το πρόβλημα της σειράς

Το `count` αναγνωρίζει κάθε instance από τη θέση του. Δες τι γίνεται όταν αφαιρέσεις το `"about"` από τη μέση της λίστας. Τη νέα λίστα τη δίνεις με το `-var` του κεφαλαίου 7 και τρέχεις μόνο plan.

TERMINAL

```
terraform plan -var 'pages=["home", "contact"]'
```

ΕΞΟΔΟΣ

```
local_file.page[0]: Refreshing state... [id=569599292125ac7f3b85df313b24dd36fcf1b835]
local_file.page[2]: Refreshing state... [id=7c558ecfc50de3b1c5e473b513eaf84b4fece7f0]
local_file.page[1]: Refreshing state... [id=c579e8f61fb6a1fb8ca32234b8c7ba5b083f98a5]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.page[1] must be replaced
-/+ resource "local_file" "page" {
      ~ content              = <<-EOT # forces replacement
          - <h1>about</h1>
          + <h1>contact</h1>
        EOT
      …
      ~ filename             = "./out/about.html" -> "./out/contact.html" # forces replacement
      ~ id                   = "c579e8f61fb6a1fb8ca32234b8c7ba5b083f98a5" -> (known after apply)
        # (2 unchanged attributes hidden)
    }

  # local_file.page[2] will be destroyed
  # (because index [2] is out of range for count)
  - resource "local_file" "page" {
      - content              = <<-EOT
            <h1>contact</h1>
        EOT -> null
      …
      - directory_permission = "0777" -> null
      - file_permission      = "0777" -> null
      - filename             = "./out/contact.html" -> null
      - id                   = "7c558ecfc50de3b1c5e473b513eaf84b4fece7f0" -> null
    }

Plan: 1 to add, 0 to change, 2 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `local_file.page[0]` εμφανίζεται μόνο στη γραμμή του refresh, γιατί η θέση `0` είναι ακόμη το `"home"`. Το `local_file.page[1]` όμως ήταν το `"about"` και τώρα η θέση `1` είναι το `"contact"`. Το `filename` και το `content` αλλάζουν και το Terraform το αντικαθιστά με `-/+`. Το `local_file.page[2]` σβήνεται με την αιτιολογία `because index [2] is out of range for count`. Η λίστα έχει πια δύο στοιχεία, οπότε δεν υπάρχει θέση `2`.

Η περίληψη λέει `1 to add` και `2 to destroy`. Ήθελες να σβήσεις ένα αρχείο και το plan αγγίζει δύο. Το `contact.html` δεν άλλαξε σε τίποτα και όμως θα σβηστεί και θα ξαναγραφτεί. Με αρχεία αυτό είναι ακίνδυνο. Με servers ή βάσεις δεδομένων σημαίνει ότι ένα αντικείμενο που δεν άγγιξες χάνεται και ξαναφτιάχνεται άδειο.



ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση δείχνει συχνά ένα resource με `count = length(var.names)` και ρωτά τι θα γίνει αν αφαιρεθεί ένα στοιχείο από τη μέση της λίστας. Τα instances κρατούν τους αριθμούς τους. Μετακινούνται τα στοιχεία, που κατεβαίνουν μία θέση. Κάθε instance μετά το στοιχείο που έφυγε παίρνει τις τιμές του επόμενου στοιχείου, οπότε αλλάζει ή αντικαθίσταται. Το τελευταίο instance σβήνεται. Η λύση που περιμένει η εξέταση είναι το `for_each`.

for_each

Το `for_each` παίρνει ένα map ή ένα set από strings και φτιάχνει ένα instance για κάθε στοιχείο. Κάθε instance αναγνωρίζεται από το **κλειδί** του και όχι από τη θέση του. Μέσα στο block υπάρχει το αντικείμενο `each` με δύο attributes.

- `each.key` είναι το κλειδί του instance.
- `each.value` είναι η τιμή. Σε set είναι ίδια με το κλειδί.

Φτιάξε δεύτερο φάκελο με την ίδια λίστα και δοκίμασε να τη δώσεις απευθείας στο `for_each`.

TERMINAL

```
cd ..
mkdir keyed
cd keyed
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "pages" {
  type    = list(string)
  default = ["home", "about", "contact"]
}

resource "local_file" "page" {
  for_each = var.pages
  filename = "${path.module}/out/${each.key}.html"
  content  = "<h1>${each.value}</h1>\n"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid for_each argument
│
│   on main.tf line 16, in resource "local_file" "page":
│   16:   for_each = var.pages
│     ├────────────────
│     │ var.pages is a list of string
│
│ The given "for_each" argument value is unsuitable: the "for_each" argument
│ must be a map, or set of strings, and you have provided a value of type
│ list of string.
╵
```

Το μήνυμα λέει ακριβώς τον κανόνα. Το `for_each` θέλει `a map, or set of strings` και πήρε `list of string`. Μια λίστα μπορεί να έχει διπλότυπα και η σειρά της μετρά, οπότε δεν δίνει μοναδικά κλειδιά. Η μετατροπή σε set με την `toset` του κεφαλαίου 10 λύνει το πρόβλημα.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "pages" {
  type    = list(string)
  default = ["home", "about", "contact"]
}

resource "local_file" "page" {
  for_each = toset(var.pages)
  filename = "${path.module}/out/${each.key}.html"
  content  = "<h1>${each.value}</h1>\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.page["about"] will be created
…
      + filename             = "./out/about.html"
…
  # local_file.page["contact"] will be created
…
      + filename             = "./out/contact.html"
…
  # local_file.page["home"] will be created
…
      + filename             = "./out/home.html"
…
Plan: 3 to add, 0 to change, 0 to destroy.
…
```

Οι διευθύνσεις έχουν τώρα κλειδί σε εισαγωγικά, όπως `local_file.page["about"]`. Εμφανίζονται ταξινομημένες αλφαβητικά, γιατί ένα set δεν έχει σειρά. Κάνε apply και αφαίρεσε ξανά το `"about"`.

TERMINAL

```
terraform apply -auto-approve
```

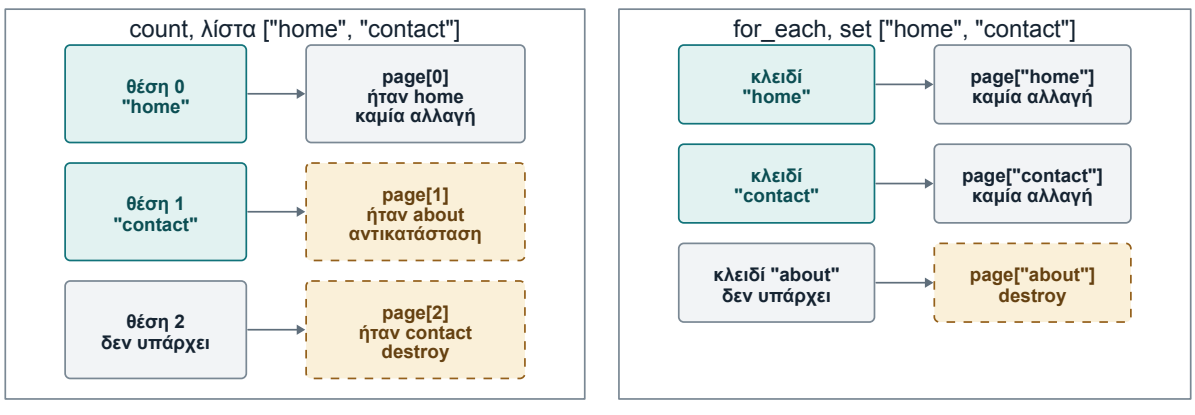
TERMINAL

```
terraform plan -var 'pages=["home", "contact"]'
```

ΕΞΟΔΟΣ

```
local_file.page["home"]: Refreshing state... [id=569599292125ac7f3b85df313b24dd36fcf1b835]
local_file.page["contact"]: Refreshing state... [id=7c558ecfc50de3b1c5e473b513eaf84b4fece7f0]
local_file.page["about"]: Refreshing state... [id=c579e8f61fb6a1fb8ca32234b8c7ba5b083f98a5]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # local_file.page["about"] will be destroyed
  # (because key ["about"] is not in for_each map)
  - resource "local_file" "page" {
      - content              = <<-EOT
            <h1>about</h1>
        EOT -> null
      …
      - directory_permission = "0777" -> null
      - file_permission      = "0777" -> null
      - filename             = "./out/about.html" -> null
      - id                   = "c579e8f61fb6a1fb8ca32234b8c7ba5b083f98a5" -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Αυτή τη φορά το plan σβήνει μόνο το `local_file.page["about"]`, με την αιτιολογία `because key ["about"] is not in for_each map`. Τα `"home"` και `"contact"` έχουν ακόμη το ίδιο κλειδί, οπότε δεν τα αγγίζει. Η περίληψη λέει `0 to add` και `1 to destroy`.



Σχήμα 11.1 · Η ίδια αλλαγή με count και με for_each. Το count ταιριάζει instances με θέσεις, οπότε όσα ακολουθούν το στοιχείο που έφυγε αλλάζουν. Το for_each ταιριάζει instances με κλειδιά, οπότε σβήνεται μόνο το κλειδί που έλειψε.

ΚΡΑΤΑ ΑΥΤΟ

Το `count` ταιριάζει κάθε instance με έναν αριθμό θέσης. Το `for_each` το ταιριάζει με ένα κλειδί. Όταν τα αντικείμενα έχουν το καθένα τη δική του ταυτότητα, όπως ένα όνομα, χρησιμοποίησε `for_each`. Το `count` ταιριάζει σε πανομοιότυπα αντίγραφα και στο «ένα ή κανένα».

ΣΗΜΕΙΩΣΗ

Ένα resource που ήδη υπάρχει με `count` δεν γυρίζει σε `for_each` με απλή αλλαγή του κώδικα. Οι διευθύνσεις αλλάζουν από `[0]` σε `["home"]` και το Terraform θα σβήσει και θα ξαναφτιάξει τα πάντα. Το κεφάλαιο 23 δείχνει πώς το block `moved` μεταφέρει τις διευθύνσεις χωρίς αντικατάσταση.

Αναφορές σε instances του for_each

Ένα resource με `for_each` είναι map από κλειδιά σε instances. Ένα instance το διαβάζεις με το κλειδί του.

TERRAFORM CONSOLE

```
> local_file.page["home"].filename
"./out/home.html"
> keys(local_file.page)
[
  "about",
  "contact",
  "home",
]
> [for p in local_file.page : p.filename]
[
  "./out/about.html",
  "./out/contact.html",
  "./out/home.html",
]
> values(local_file.page)[*].filename
[
  "./out/about.html",
  "./out/contact.html",
  "./out/home.html",
]
```

Η `keys` δίνει τα κλειδιά. Για όλα τα `filename` γράφεις μια έκφραση `for` ή παίρνεις πρώτα τις τιμές με τη `values` και μετά κάνεις splat. Το splat απευθείας πάνω στο resource και ο αριθμός ως index δεν δουλεύουν.

TERRAFORM CONSOLE

```
> local_file.page[*].filename
╷
│ Error: Unsupported attribute
│
│   on <console-input> line 1:
│   (source code not available)
│
│ This object does not have an attribute named "filename".
╵
> local_file.page[0]
╷
│ Error: Invalid index
│
│   on <console-input> line 1:
│   (source code not available)
│
│ The given key does not identify an element in this collection value. An
│ object only supports looking up attributes by name, not by numeric index.
╵
```

Το splat δουλεύει σε λίστες, όπως είδες στο κεφάλαιο 10. Πάνω σε map δεν βρίσκει attribute `filename` και σταματά με `Unsupported attribute`. Το `[0]` αποτυγχάνει γιατί το resource δεν έχει θέσεις, μόνο κλειδιά. Το μήνυμα μιλά για object, που στην πρόσβαση δουλεύει όπως το map. Εξηγεί ότι δέχεται μόνο ονόματα, `not by numeric index`.

for_each με map και object

Με map το `each.key` είναι το κλειδί και το `each.value` η τιμή, που μπορεί να είναι και object. Έτσι κάθε instance παίρνει όσες ρυθμίσεις χρειάζεται από μία μεταβλητή. Φτιάξε τον φάκελο `users`.

TERMINAL

```
cd ..
mkdir users
cd users
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "users" {
  type = map(object({
    role  = string
    admin = optional(bool, false)
  }))
  default = {
    anna  = { role = "developer", admin = true }
    nikos = { role = "analyst" }
    eleni = { role = "developer" }
  }
}

resource "local_file" "user" {
  for_each = var.users
  filename = "${path.module}/users/${each.key}.txt"
  content  = "${each.key} ${each.value.role}\n"
}

resource "local_file" "admin" {
  for_each = { for name, u in var.users : name => u if u.admin }
  filename = "${path.module}/admins/${each.key}.txt"
  content  = "${each.key}\n"
}
```

Διάβασε τα δύο resources.

- Το `local_file.user` παίρνει το map όπως είναι. Το `each.key` είναι το όνομα του χρήστη και το `each.value` το object με τα `role` και `admin`. Το `each.value.role` διαβάζει ένα attribute του object.

- Το `local_file.admin` παίρνει το αποτέλεσμα μιας έκφρασης `for` με άγκιστρα, που δίνει object. Το `if u.admin` κρατά μόνο όσους έχουν `admin = true`. Ο Νίκος και η Ελένη δεν το έγραψαν, οπότε το `optional` του κεφαλαίου 9 τους έδωσε `false`.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.admin["anna"] will be created
…
  # local_file.user["anna"] will be created
…
  # local_file.user["eleni"] will be created
…
  # local_file.user["nikos"] will be created
…
Plan: 4 to add, 0 to change, 0 to destroy.
…
```

Το `local_file.user` έχει τρία instances, ένα για κάθε κλειδί. Το `local_file.admin` έχει ένα, μόνο για την `anna`. Το `for_each` δέχεται οποιαδήποτε έκφραση δίνει map ή set, όχι μόνο μια μεταβλητή.

Συχνά τα δεδομένα έρχονται ως λίστα από objects, όπως η `var.users` του κεφαλαίου 10. Η λίστα δεν περνά στο `for_each`, αλλά μια έκφραση `for` τη μετατρέπει σε map με κλειδί ένα μοναδικό attribute.

HCL

```
resource "local_file" "member" {
  for_each = { for u in var.members : u.name => u }
  filename = "${path.module}/members/${each.key}.txt"
  content  = "${each.value.team}\n"
}
```

ΠΑΓΙΔΑ

Το κλειδί πρέπει να είναι μοναδικό. Αν δύο objects έχουν το ίδιο `name`, η έκφραση `for` σταματά με `Duplicate object key`, όπως είδες στο κεφάλαιο 10. Διάλεξε ως κλειδί κάτι σταθερό που δεν θα αλλάξει. Αν αλλάξει το κλειδί, το Terraform σβήνει το παλιό instance και φτιάχνει νέο.

Τα κλειδιά πρέπει να είναι γνωστά στο plan

Το Terraform πρέπει να ξέρει στο plan πόσα instances θα υπάρξουν και ποια είναι η διεύθυνση του καθενός. Γι' αυτό τα κλειδιά του `for_each` και ο αριθμός του `count` δεν μπορούν να εξαρτώνται από τιμές που είναι `(known after apply)`. Δοκίμασε κλειδιά από τυχαία ονόματα του `random_pet` του κεφαλαίου 6.

TERMINAL

```
cd ..
mkdir pets
cd pets
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "name" {
  count = 2
}

resource "local_file" "pet" {
  for_each = toset(random_pet.name[*].id)
  filename = "${path.module}/out/${each.key}.txt"
  content  = "${each.key}\n"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
╷
│ Error: Invalid for_each argument
│
│   on main.tf line 19, in resource "local_file" "pet":
│   19:   for_each = toset(random_pet.name[*].id)
│      ├────────────────
│      │ random_pet.name is tuple with 2 elements
│
│ The "for_each" set includes values derived from resource attributes that
│ cannot be determined until apply, and so Terraform cannot determine the
│ full set of keys that will identify the instances of this resource.
│
│ When working with unknown values in for_each, it's better to use a map
│ value where the keys are defined statically in your configuration and where
│ only the values contain apply-time results.
```

```
Alternatively, you could use the -target planning option to first apply
only the resources that the for_each value depends on, and then apply a
second time to fully converge.
```

Τα ονόματα των pets θα υπάρξουν μόνο στο apply, οπότε το Terraform δεν ξέρει τι διευθύνσεις να δώσει στα αρχεία. Το μήνυμα προτείνει δύο δρόμους. Ο σωστός είναι ένα map με κλειδιά γραμμένα σταθερά στη configuration, όπου μόνο οι τιμές έρχονται από το apply. Ο δεύτερος είναι το `-target` του κεφαλαίου 3 και δύο διαδοχικά apply, που είναι παράκαμψη και όχι λύση.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "name" {
  for_each = toset(["web", "db"])
  prefix   = each.key
}

resource "local_file" "pet" {
  for_each = random_pet.name
  filename = "${path.module}/out/${each.key}.txt"
  content  = "${each.value.id}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.pet["db"] will be created
…
      + content              = (known after apply)
…
  # local_file.pet["web"] will be created
…
      + content              = (known after apply)
…
  # random_pet.name["db"] will be created
…
  # random_pet.name["web"] will be created
…
Plan: 4 to add, 0 to change, 0 to destroy.
…
```

Το `local_file.pet` παίρνει ως `for_each` ολόκληρο το `random_pet.name`. Είναι map με κλειδιά `"db"` και `"web"`, γνωστά από τη configuration. Το `each.value` είναι ένα ολόκληρο instance του `random_pet`, οπότε το `each.value.id` είναι το τυχαίο όνομα. Η τιμή του είναι ακόμη `(known after apply)`, κάτι που επιτρέπεται για τιμές.

Δημιουργία υπό όρους

Το Terraform δεν έχει block `if` για resources. Το ίδιο αποτέλεσμα το δίνει ένα `count` με έκφραση υπό όρους. Με `1` το resource υπάρχει, με `0` όχι.

TERMINAL

```
cd ..
mkdir toggle
cd toggle
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "debug" {
  type    = bool
  default = false
}

resource "local_file" "debug" {
  count    = var.debug ? 1 : 0
  filename = "${path.module}/debug.log"
  content  = "debug on\n"
}

output "debug_file" {
  value = local_file.debug[0].filename
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Invalid index
│
│   on main.tf line 22, in output "debug_file":
│   22:   value = local_file.debug[0].filename
│      ├────────────────
│      │ local_file.debug is empty tuple
│
│ The given key does not identify an element in this collection value: the
│ collection has no elements.
╵
```

Με `debug = false` το `local_file.debug` είναι άδεια λίστα, `empty tuple` όπως λέει το μήνυμα. Το `[0]` δεν υπάρχει και το plan σταματά. Ένα resource που μπορεί να μην υπάρχει θέλει αναφορά που αντέχει την απουσία του. Υπάρχουν δύο τρόποι.

- Η συνάρτηση `one` δέχεται λίστα με κανένα ή ένα στοιχείο. Με κανένα επιστρέφει `null` και με ένα επιστρέφει το στοιχείο. Με περισσότερα βγάζει σφάλμα, που εδώ είναι προστασία. Το `one(local_file.debug[*].filename)` είναι ο συνηθισμένος τρόπος.
- Η `try` του κεφαλαίου 10 δοκιμάζει το `[0]` και, αν αποτύχει, δίνει μια προεπιλογή.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "debug" {
  type    = bool
  default = false
}

resource "local_file" "debug" {
  count    = var.debug ? 1 : 0
  filename = "${path.module}/debug.log"
  content  = "debug on\n"
}

output "debug_file" {
  value = one(local_file.debug[*].filename)
}

output "debug_try" {
  value = try(local_file.debug[0].filename, "κανένα")
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

debug_try = "κανένα"
```

Χωρίς debug δεν φτιάχτηκε τίποτα. Το `debug_try` πήρε την προεπιλογή. Το `debug_file` δεν εμφανίζεται καθόλου, γιατί ένα output με τιμή `null` παραλείπεται. Άνοιξε τώρα τον διακόπτη.

TERMINAL

```
terraform apply -auto-approve -var debug=true
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

debug_file = "./debug.log"
debug_try = "./debug.log"
```

Το αρχείο φτιάχτηκε και τα δύο outputs δείχνουν το path του.

ΠΑΓΙΔΑ

Μη γράφεις σκέτο `[0]` σε resource με `count` υπό όρους. Δουλεύει όσο ο διακόπτης είναι ανοιχτός και σπάει το plan τη μέρα που κάποιος θα τον κλείσει. Χρησιμοποίησε `one(...[*]...)` ή `try`.

Το ίδιο μοτίβο γράφεται και με `for_each`, με ένα άδειο map όταν η συνθήκη είναι `false`. Για «ένα ή κανένα» το `count` είναι πιο σύντομο και πιο συνηθισμένο.

count και for_each μαζί

Ένα block παίρνει είτε `count` είτε `for_each`, ποτέ και τα δύο. Πρόσθεσε στον φάκελο ένα αρχείο που τα βάζει μαζί.

wrong.tf

```
resource "local_file" "both" {
  count    = 2
  for_each = toset(["a", "b"])
  filename = "${path.module}/${each.key}.txt"
  content  = "${each.key}\n"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid combination of "count" and "for_each"
│
│   on wrong.tf line 3, in resource "local_file" "both":
│    3:   for_each = toset(["a", "b"])
│
│ The "count" and "for_each" meta-arguments are mutually-exclusive, only one
│ should be used to be explicit about the number of resources to be created.
╵
```

Το σφάλμα λέει ότι τα δύο meta-arguments είναι `mutually-exclusive`. Το βρίσκει ήδη το `validate`, χωρίς plan. Ένα δεύτερο λάθος της ίδιας οικογένειας είναι το `count.index` σε block χωρίς `count`.

wrong.tf

```
resource "local_file" "lonely" {
  filename = "${path.module}/lonely.txt"
  content  = "${count.index}\n"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Reference to "count" in non-counted context
│
│   on wrong.tf line 3, in resource "local_file" "lonely":
│    3:   content  = "${count.index}\n"
│
│ The "count" object can only be used in "module", "resource", and "data"
│ blocks, and only when the "count" argument is set.
╵
```

Το `count` υπάρχει μόνο σε block που ορίζει `count`. Με τον ίδιο τρόπο το `each` υπάρχει μόνο σε block με `for_each`. Σβήσε το αρχείο.

TERMINAL

```
rm wrong.tf
```

dynamic blocks

Ορισμένα resources έχουν **nested blocks**, blocks μέσα στο block του resource που γράφονται όσες φορές χρειάζεται. Ένα τέτοιο είναι το `archive_file` του provider `hashicorp/archive`, που φτιάχνει αρχεία zip στον δίσκο σου. Κάθε αρχείο μέσα στο zip είναι ένα block `source` με το όνομά του και το περιεχόμενό του.



TERMINAL

```
cd ..
mkdir zip
cd zip
```

main.tf

```
terraform {
  required_providers {
    archive = {
      source  = "hashicorp/archive"
      version = "2.8.1"
    }
  }
}

resource "archive_file" "site" {
  type        = "zip"
  output_path = "${path.module}/site.zip"

  source {
    filename = "index.html"
    content  = "<h1>Αρχική</h1>\n"
  }

  source {
    filename = "about.html"
    content  = "<h1>Ποιοι είμαστε</h1>\n"
  }
}
```

Το `type` λέει τι είδους αρχείο θα φτιαχτεί και το `output_path` πού θα γραφτεί. Τα δύο blocks `source` διαφέρουν μόνο στις τιμές τους. Αν οι σελίδες ζουν σε μια μεταβλητή, θέλεις ένα `source` για κάθε στοιχείο της. Το `count` και το `for_each` δεν βοηθούν εδώ, γιατί πολλαπλασιάζουν όλο το resource και θα έφτιαχναν πολλά zip. Για nested blocks υπάρχει το block `dynamic`.

main.tf

```
terraform {
  required_providers {
    archive = {
      source  = "hashicorp/archive"
      version = "2.8.1"
    }
  }
}

variable "pages" {
  type = map(string)
  default = {
    "index.html" = "Αρχική"
    "about.html" = "Ποιοι είμαστε"
  }
}

resource "archive_file" "site" {
  type        = "zip"
  output_path = "${path.module}/site.zip"

  dynamic "source" {
    for_each = var.pages
    content {
      filename = source.key
      content  = "<h1>${source.value}</h1>\n"
    }
  }
}
```

Διάβασε το block `dynamic` γραμμή προς γραμμή.

- Το label `"source"` είναι ο τύπος του nested block που θα παραχθεί.
- Το `for_each` δίνει τη συλλογή. Δέχεται map, set, list ή tuple. Εδώ κάθε κλειδί του map είναι όνομα αρχείου και κάθε τιμή ένας τίτλος.
- Το `content` είναι το σώμα κάθε παραγόμενου block. Ό,τι γράφεις μέσα του γίνεται τα arguments ενός `source`. Μην το μπερδέψεις με το argument `content` του ίδιου του `source`, που βρίσκεται μέσα του και απλώς έχει το ίδιο όνομα.
- Μέσα στο `content` η τρέχουσα τιμή λέγεται όπως το label, εδώ `source`. Το `source.key` είναι το κλειδί και το `source.value` η τιμή. Σε λίστα το `key` είναι η θέση και το `value` το στοιχείο.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
```

```
  + create

Terraform will perform the following actions:

  # archive_file.site will be created
  + resource "archive_file" "site" {
      + id                  = (known after apply)
      …
      + output_path         = "./site.zip"
      …
      + type                = "zip"

      + source {
          + content  = <<-EOT
                <h1>Αρχική</h1>
            EOT
          + filename = "index.html"
        }
      + source {
          + content  = <<-EOT
                <h1>Ποιοι είμαστε</h1>
            EOT
          + filename = "about.html"
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το plan δείχνει δύο blocks `source` ακριβώς σαν να τα είχες γράψει με το χέρι. Το `dynamic` υπάρχει μόνο στον κώδικα. Ο provider βλέπει απλά nested blocks.

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
unzip -l site.zip
```

ΕΞΟΔΟΣ

```
Archive:  site.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
       35  00-00-1980 00:00   about.html
       22  00-00-1980 00:00   index.html
---------                     -------
       57                     2 files
```

Το zip έχει δύο αρχεία, ένα για κάθε κλειδί του map. Η ημερομηνία `00-00-1980` δεν είναι λάθος. Ο provider γράφει σταθερή ημερομηνία, ώστε το ίδιο περιεχόμενο να δίνει πάντα το ίδιο zip με το ίδιο checksum. Αν δεν έχεις `unzip`, το `ls -l` δείχνει τουλάχιστον ότι το `site.zip` υπάρχει.

iterator

Το όνομα `source` για την τρέχουσα τιμή μπερδεύει όταν το nested block έχει γενικό όνομα. Το argument `iterator` του δίνει όποιο όνομα θέλεις.

main.tf

```
terraform {
  required_providers {
    archive = {
      source  = "hashicorp/archive"
      version = "2.8.1"
    }
  }
}

variable "pages" {
  type = map(string)
  default = {
    "index.html" = "Αρχική"
    "about.html" = "Ποιοι είμαστε"
  }
}

resource "archive_file" "site" {
  type        = "zip"
  output_path = "${path.module}/site.zip"

  dynamic "source" {
    for_each = var.pages
    iterator = page
    content {
      filename = page.key
      content  = "<h1>${page.value}</h1>\n"
    }
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
archive_file.site: Refreshing state... [id=fa93dbcb4c7438936a7e382ae4df0288fb5067cb]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το `No changes` δείχνει ότι το `iterator` αλλάζει μόνο το όνομα μέσα στον κώδικα. Τα blocks που παράγονται μένουν ίδια.

Μόνο για blocks

Το `dynamic` παράγει nested blocks. Δεν παράγει arguments. Το `type` και το `output_path` είναι arguments, οπότε ένα `dynamic "type"` δεν έχει νόημα.

wrong.tf

```
resource "archive_file" "bad" {
  output_path = "${path.module}/bad.zip"

  dynamic "type" {
    for_each = ["zip"]
    content {
      value = type.value
    }
  }
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Missing required argument
│
│   on wrong.tf line 1, in resource "archive_file" "bad":
│    1: resource "archive_file" "bad" {
│
│ The argument "type" is required, but no definition was found.
╵

╷
│ Error: Unsupported block type
│
│   on wrong.tf line 4, in resource "archive_file" "bad":
│    4:   dynamic "type" {
│
│ Blocks of type "type" are not expected here.
╵
```

Το `validate` απαντά ότι λείπει το υποχρεωτικό argument `type` και ότι `Blocks of type "type" are not expected here`. Για ένα argument που αλλάζει αρκεί μια έκφραση, για παράδειγμα υπό όρους.

TERMINAL

```
rm wrong.tf
```

ΠΑΓΙΔΑ

Το `dynamic` κάνει τον κώδικα πιο δύσκολο να διαβαστεί. Όποιος τον ανοίξει πρέπει να υπολογίσει στο μυαλό του ποια blocks θα βγουν. Για δύο ή τρία σταθερά blocks γράψε τα στατικά, όπως στην πρώτη εκδοχή του `site.zip`. Χρησιμοποίησε `dynamic` όταν τα blocks προέρχονται από δεδομένα με πλήθος που αλλάζει, όπως μια μεταβλητή ή ένα input ενός module.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Για το `dynamic` η εξέταση ελέγχει τρία σημεία. Παράγει επαναλαμβανόμενα nested blocks μέσα σε resource, όχι arguments και όχι ολόκληρα resources. Το σώμα κάθε block γράφεται μέσα σε `content`. Η τρέχουσα τιμή λέγεται όπως το label του block, εκτός αν ορίσεις άλλο όνομα με `iterator`.

count ή for_each

	count	for_each
Δέχεται	ακέραιο αριθμό	map ή set από strings
Μέσα στο block	`count.index`	`each.key`, `each.value`
Διεύθυνση	`local_file.page[0]`	`local_file.page["home"]`
Το resource είναι	λίστα	map
Αφαίρεση στοιχείου	αλλάζει όσα ακολουθούν, σβήνει το τελευταίο	σβήνει μόνο αυτό
Ταιριάζει σε	αντίγραφα, «ένα ή κανένα»	αντικείμενα με ταυτότητα

Και τα δύο θέλουν τιμή γνωστή στο plan και δεν συνυπάρχουν στο ίδιο block. Δουλεύουν με τον ίδιο τρόπο και σε modules, όπως θα δεις στο κεφάλαιο 15.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch11`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από υπάρχοντα. Γράψε το `main.tf`, τρέξε `terraform init` και μετά τις εντολές που ζητά η εκφώνηση.

ΑΣΚΗΣΗ 11.1 Τρία αντίγραφα

Γράψε ένα resource `local_file` με όνομα `server` και `count = 3`. Τα αρχεία λέγονται `server-1.txt`, `server-2.txt` και `server-3.txt`, δηλαδή η αρίθμηση ξεκινά από το 1. Κάθε αρχείο περιέχει τον αριθμό του και αλλαγή γραμμής. Ένα output με όνομα `files` δίνει όλα τα `filename` με splat. Κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

files = [
  "./server-1.txt",
  "./server-2.txt",
  "./server-3.txt",
]
```

ΑΣΚΗΣΗ 11.2 Διπλότυπα σε set

Η μεταβλητή `envs` είναι `list(string)` με default `["dev", "prod", "dev", "test"]`. Γράψε ένα `local_file` με όνομα `env` και `for_each`, που φτιάχνει ένα αρχείο `ΠΕΡΙΒΑΛΛΟΝ.env` για κάθε περιβάλλον με περιεχόμενο `ENVIRONMENT=ΠΕΡΙΒΑΛΛΟΝ` και αλλαγή γραμμής. Κάνε apply και δείξε τις διευθύνσεις με `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
local_file.env["dev"]
local_file.env["prod"]
local_file.env["test"]
```

ΑΣΚΗΣΗ 11.3 Πρόβλεψε το plan

Ξεκίνα από αντίγραφο του φακέλου `pages`, με το `count` πάνω στη λίστα σελίδων. Κάνε apply με τις προεπιλογές. Πριν τρέξεις plan, γράψε σε χαρτί τι θα πει η περίληψη αν προσθέσεις τη σελίδα `"blog"` στην αρχή της λίστας. Έλεγξε με `terraform plan -var` και τη λίστα `["blog", "home", "about", "contact"]`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # local_file.page[0] must be replaced
…
  # local_file.page[1] must be replaced
…
  # local_file.page[2] must be replaced
…
  # local_file.page[3] will be created
…
Plan: 4 to add, 0 to change, 3 to destroy.
…
```

ΑΣΚΗΣΗ 11.4 Αρχείο υπό όρους

Γράψε configuration με μεταβλητή `readme` τύπου `bool` και default `false`. Ένα `local_file` με όνομα `readme` φτιάχνει το `README.txt` μόνο όταν η μεταβλητή είναι `true`. Ένα output με όνομα `readme_path` δίνει το `filename` του ή `null`, χωρίς να σπάει όταν το αρχείο δεν υπάρχει. Κάνε apply με `-var readme=true` και μετά ξανά χωρίς το `-var`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # local_file.readme[0] will be destroyed
…
  - readme_path = "./README.txt" -> null
…
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.
```

ΑΣΚΗΣΗ 11.5 Χρήστες από λίστα objects

Η μεταβλητή `members` είναι λίστα από objects με `name`, `team` και `active`, όπου το `active` είναι `optional(bool, true)`. Δώσε default με τρία μέλη, την `anna` στην ομάδα `web`, τον `nikos` στην ομάδα `data` με `active = false` και την `maria` στην ομάδα `data`. Με ένα `local_file` με όνομα `member` φτιάξε ένα αρχείο `members/ONOMA.txt` με περιεχόμενο την ομάδα, μόνο για τα ενεργά μέλη. Οι διευθύνσεις πρέπει να έχουν ως κλειδί το όνομα. Κάνε apply και δείξε τις διευθύνσεις με `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
local_file.member["anna"]
local_file.member["maria"]
```

ΑΣΚΗΣΗ 11.6 Σημειώσεις σε zip

Η μεταβλητή `notes` είναι λίστα από objects με `file` και `text`, με default δύο σημειώσεις, το `todo.txt` με κείμενο `να γράψω tests` και το `ideas.txt` με κείμενο `ένα module για όλα`. Φτιάξε ένα `archive_file` με όνομα `notes` που γράφει το `notes.zip` με ένα block `source` ανά σημείωση. Χρησιμοποίησε `dynamic` με `iterator = note`. Κάθε αρχείο τελειώνει με αλλαγή γραμμής. Κάνε apply και δείξε τα περιεχόμενα του zip με `unzip -l`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Archive:  notes.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
       28  00-00-1980 00:00   ideas.txt
       22  00-00-1980 00:00   todo.txt
---------                     -------
       50                     2 files
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 11.1

Ένα resource έχει `count = length(var.names)` με `names = ["a", "b", "c"]`. Έγινε apply. Αφαιρείς το `"b"` και τρέχεις `terraform plan`. Τι δείχνει το plan;

a. Σβήνει μόνο το instance του `"b"`.
b. Καμία αλλαγή, γιατί τα `"a"` και `"c"` υπάρχουν ακόμη.
c. Αλλάζει ή αντικαθιστά το instance `[1]` ώστε να γίνει `"c"` και σβήνει το `[2]`.
d. Σβήνει και τα τρία instances και φτιάχνει δύο νέα.

ΕΡΩΤΗΣΗ 11.2

Ποιους τύπους τιμής δέχεται το meta-argument `for_each` σε ένα resource; Επίλεξε δύο.

a. `list(string)`
b. `map` με οποιονδήποτε τύπο τιμών
c. `number`
d. `set(string)`
e. `tuple`

ΕΡΩΤΗΣΗ 11.3

Το resource `random_pet` με όνομα `name` έχει `for_each = toset(["web", "db"])`. Ποια είναι η διεύθυνση του instance για το `"db"`;

a. `random_pet.name["db"]`
b. `random_pet.name[1]`
c. `random_pet.name.db`
d. `random_pet.db`

ΕΡΩΤΗΣΗ 11.4

Σωστό ή λάθος; Ένα block `resource` μπορεί να έχει ταυτόχρονα `count` και `for_each`, για παράδειγμα δύο αντίγραφα για κάθε κλειδί.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 11.5

Το resource `local_file.x` έχει `count = var.create ? 1 : 0`. Ένα output πρέπει να δίνει το `id` του όταν υπάρχει και `null` όταν δεν υπάρχει. Ποια έκφραση το κάνει χωρίς σφάλμα και στις δύο περιπτώσεις;

a. `local_file.x.id`
b. `local_file.x[0].id`
c. `local_file.x["0"].id`
d. `one(local_file.x[*].id)`

ΕΡΩΤΗΣΗ 11.6

Ποια πρόταση για τα blocks `dynamic` είναι σωστή;

a. Παράγουν arguments με τιμές από μια συλλογή, για παράδειγμα πολλά `filename`.
b. Αντικαθιστούν το `for_each` όταν θέλεις πολλά instances ενός resource.
c. Παράγουν επαναλαμβανόμενα nested blocks και η τρέχουσα τιμή λέγεται όπως το label, εκτός αν οριστεί `iterator`.
d. Επιτρέπονται μόνο μέσα σε modules.



ΚΕΦΑΛΑΙΟ 12

Data sources

Μέχρι τώρα κάθε block της configuration περιέγραφε κάτι που φτιάχνει το Terraform. Συχνά όμως χρειάζεσαι πληροφορία που υπάρχει ήδη, για παράδειγμα ένα αρχείο ρυθμίσεων που γράφει άλλη ομάδα, το image ενός server ή τα outputs μιας άλλης configuration. Σε αυτό το κεφάλαιο μαθαίνεις το block `data`, που διαβάζει τέτοια πληροφορία χωρίς να τη διαχειρίζεται. Βλέπεις επίσης πότε ακριβώς γίνεται η ανάγνωση.

Διαχείριση και ανάγνωση

Ένα block `resource` κάνει το Terraform ιδιοκτήτη ενός αντικειμένου. Το Terraform το φτιάχνει στο πρώτο apply, το αλλάζει όταν αλλάζει η configuration και το σβήνει στο destroy. Αυτός είναι ο κύκλος ζωής που ακολουθείς από το κεφάλαιο 1 και τον καταγράφει το state.

Ένα **data source** κάνει μόνο ένα πράγμα. Ζητά από τον provider πληροφορία για κάτι που υπάρχει ήδη και φέρνει τις τιμές της στη configuration. Δεν το φτιάχνει, δεν το αλλάζει και δεν το σβήνει. Το αντικείμενο μπορεί να το έφτιαξε ένας άνθρωπος με το χέρι, ένα script, μια άλλη configuration ή ο ίδιος ο πάροχος. Τα images που δημοσιεύουν το AWS και οι διανομές Linux σε κάθε region είναι το κλασικό παράδειγμα.

Το block γράφεται όπως το `resource`, με τη λέξη `data` στη θέση του.

```hcl
data "local_file" "shift" {
  filename = "shift.txt"
}
```

Το πρώτο label είναι ο τύπος του data source και το δεύτερο το όνομα που του δίνεις. Τα arguments είναι τα κριτήρια της αναζήτησης, εδώ ποιο αρχείο θα διαβαστεί. Τα attributes είναι το αποτέλεσμα, για παράδειγμα το περιεχόμενο του αρχείου. Ποια data sources υπάρχουν και τι δέχεται το καθένα το ορίζει ο provider, όπως και για τα resources. Ο `hashicorp/local` έχει τον τύπο `local_file` και ως resource και ως data source. Το resource γράφει ένα αρχείο. Το data source διαβάζει ένα που υπάρχει ήδη.

Στα μηνύματά του το Terraform λέει τα resources **managed resources** και τα data sources **data resources**. Θα συναντήσεις και τους δύο όρους στην έξοδο.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Ένα `resource` διαχειρίζεται τον κύκλο ζωής ενός αντικειμένου. Ένα `data` μόνο διαβάζει. Ό,τι διαβάζει ένα data source, το Terraform δεν το φτιάχνει, δεν το αλλάζει και δεν το σβήνει.

Το πρώτο data source

Φτιάξε τον φάκελο του παραδείγματος. Το αρχείο που θα διαβάσεις το γράφει κάποιος άλλος, εδώ εσύ με `echo`. Για το Terraform είναι ένα αρχείο που απλώς υπάρχει.

TERMINAL

```
mkdir notes
cd notes
echo "Η σημερινή βάρδια: Άννα" > shift.txt
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

data "local_file" "shift" {
  filename = "${path.module}/shift.txt"
}

output "shift" {
  value = trimspace(data.local_file.shift.content)
}
```

Το output διαβάζει το attribute `content` του data source. Η αναφορά ξεκινά με `data` και συνεχίζει με τον τύπο, το όνομα και το attribute. Το `echo` βάζει αλλαγή γραμμής στο τέλος του αρχείου και η `trimspace` του κεφαλαίου 10 την κόβει.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
data.local_file.shift: Reading...
data.local_file.shift: Read complete after 0s [id=b9b26b80356041152b1d4d1caa86a06b93f0a935]

Changes to Outputs:
  + shift = "Η σημερινή βάρδια: Άννα"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Οι δύο πρώτες γραμμές είναι η ανάγνωση. Το `Reading...` λέει ότι ξεκίνησε και το `Read complete` ότι τελείωσε. Σε αγκύλες είναι το `id`, που για τον provider `local` είναι το SHA1 του περιεχομένου, όπως στο resource του κεφαλαίου 1. Η ανάγνωση έγινε μέσα στο plan, πριν από τον υπολογισμό των αλλαγών. Γι' αυτό το output έχει ήδη την τελική του τιμή.

Λείπει η γραμμή `Plan:`, γιατί δεν υπάρχει resource για να φτιαχτεί. Το Terraform λέει ότι το apply θα αποθηκεύσει μόνο τις νέες τιμές των outputs, χωρίς να αλλάξει καμία πραγματική υποδομή.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.local_file.shift: Reading...
data.local_file.shift: Read complete after 0s [id=b9b26b80356041152b1d4d1caa86a06b93f0a935]

Changes to Outputs:
  + shift = "Η σημερινή βάρδια: Άννα"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

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

Outputs:

shift = "Η σημερινή βάρδια: Άννα"
```

Το apply διάβασε ξανά το αρχείο και έγραψε το output. Η περίληψη λέει `0 added`. Ένα data source δεν μετρά ποτέ στα resources που προστίθενται, αλλάζουν ή σβήνονται.

Η διεύθυνση και το state

Η διεύθυνση ενός data source είναι `data.τύπος.όνομα`, εδώ `data.local_file.shift`. Το πρόθεμα `data` το ξεχωρίζει από τα resources. Τα `local_file.shift` και `data.local_file.shift` είναι δύο διαφορετικές

διευθύνσεις και μπορούν να υπάρχουν μαζί στην ίδια configuration. Ένα attribute το διαβάζεις όπως στα resources, με τελεία και το όνομά του.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
data.local_file.shift
```

Το apply έγραψε το data source στο state, παρόλο που δεν έφτιαξε τίποτα. Εκεί μένει η τιμή της τελευταίας ανάγνωσης.

TERMINAL

```
terraform state show data.local_file.shift
```

ΕΞΟΔΟΣ

```
# data.local_file.shift:
data "local_file" "shift" {
    content              = <<-EOT
        Η σημερινή βάρδια: Άννα
    EOT
    content_base64       = "zpcgz4POt868zrXPgc65zr3OriDOss6sz4HOtM65zrE6IM6Gzr3Ovc6xCg=="
    …
    filename             = "./shift.txt"
    id                   = "b9b26b80356041152b1d4d1caa86a06b93f0a935"
}
```

Η πρώτη γραμμή είναι η διεύθυνση και το block γράφει `data` αντί για `resource`. Το `filename` είναι το argument που έδωσες. Τα υπόλοιπα τα συμπλήρωσε ο provider. Το `content` είναι το κείμενο του αρχείου και το `content_base64` το ίδιο κείμενο σε base64. Οι γραμμές με τα checksums κόβονται για συντομία.

Ένα λάθος που γίνεται συχνά είναι να ξεχάσεις το `data` στην αναφορά. Πρόσθεσε για λίγο ένα δεύτερο output που το παραλείπει.

typo.tf

```
output "typo" {
  value = local_file.shift.content
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Reference to undeclared resource
│
│   on typo.tf line 2, in output "typo":
│    2:   value = local_file.shift.content
│
│ A managed resource "local_file" "shift" has not been declared in the root
│ module.
│
│ Did you mean the data resource data.local_file.shift?
╵
```

Το Terraform ψάχνει managed resource `local_file.shift`, δεν το βρίσκει και προτείνει το data resource με το ίδιο όνομα. Σβήσε το αρχείο.

TERMINAL

```
rm typo.tf
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Μια αναφορά σε data source ξεκινά πάντα με `data.`, όπως `data.local_file.shift.content` ή `data.aws_ami.ubuntu.id`. Χωρίς το πρόθεμα η αναφορά δείχνει σε resource. Η εξέταση βάζει συχνά τις δύο μορφές δίπλα δίπλα στις επιλογές.

Κάθε plan διαβάζει ξανά

Το state κρατά την τελευταία ανάγνωση, αλλά το Terraform δεν βασίζεται σε αυτήν. Σε κάθε plan, apply και destroy το data source διαβάζεται από την αρχή. Άλλαξε το αρχείο έξω από το Terraform.

TERMINAL

```
echo "Η σημερινή βάρδια: Νίκος" > shift.txt
terraform plan
```

ΕΞΟΔΟΣ

```
data.local_file.shift: Reading...
data.local_file.shift: Read complete after 0s [id=88017672d772eb33b4352290db2062830d58d615]

Changes to Outputs:
  ~ shift = "Η σημερινή βάρδια: Άννα" -> "Η σημερινή βάρδια: Νίκος"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `id` είναι άλλο, γιατί άλλαξε το περιεχόμενο. Το output δείχνει την παλιά και τη νέα τιμή με `~`. Δεν υπάρχει τίποτα να διορθωθεί. Σε ένα resource μια αλλαγή έξω από το Terraform είναι drift και το plan

προτείνει να την αναιρέσει, όπως είδες στο κεφάλαιο 4 με το `rules.txt`. Σε ένα data source ο έξω κόσμος έχει πάντα δίκιο. Το Terraform απλώς φέρνει τη νέα τιμή σε όσα την αναφέρουν.

Τι γίνεται αν το αρχείο λείπει; Σβήσε το και τρέξε ξανά plan.

TERMINAL

```
rm shift.txt
terraform plan
```

ΕΞΟΔΟΣ

```
data.local_file.shift: Reading...

Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Read local file data source error
│
│   with data.local_file.shift,
│   on main.tf line 10, in data "local_file" "shift":
│   10: data "local_file" "shift" {
│
│ The file at given path cannot be read.
│
│ +Original Error: open ./shift.txt: no such file or directory
╵
```

Το plan σταμάτησε στην ανάγνωση. Το μήνυμα `Read local file data source error` είναι του provider και η γραμμή `Original Error` λέει την αιτία, `no such file or directory`. Ένα data source που δεν βρίσκει αυτό που ζητά δεν δίνει κενή τιμή. Σταματά όλο το plan. Το ίδιο κάνουν και τα data sources του cloud που ψάχνουν ένα συγκεκριμένο αντικείμενο, όπως ένα image. Έτσι το plan δεν προχωρά ποτέ με τιμή που δεν βρέθηκε.

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

Το `validate` πέρασε. Ελέγχει τη σύνταξη και τα arguments με βάση το schema του provider, αλλά δεν ζητά από τον provider να διαβάσει τίποτα. Ένα αρχείο που λείπει φαίνεται μόνο στο plan.

Data source ή συνάρτηση file

Στο κεφάλαιο 10 διάβασες αρχεία με τη συνάρτηση `file`. Οι δύο τρόποι μοιάζουν, αλλά διαφέρουν σε τρία σημεία.

- **Χρόνος.** Η `file` εκτελείται όταν υπολογίζεται η έκφραση και το αρχείο πρέπει να υπάρχει ήδη. Ένα data source είναι κόμβος στο graph των εξαρτήσεων του κεφαλαίου 6. Μπορεί να περιμένει ένα resource που φτιάχνει το αρχείο, όπως θα δεις παρακάτω.
- **State.** Το data source γράφει ό,τι διάβασε στο state. Η `file` δεν αφήνει ίχνος.

- **Attributes.** Το data source δίνει επίσης το `content_base64` και έτοιμα checksums, όπως το `content_sha256`.

Για ένα αρχείο που ζει μαζί με τον κώδικα, όπως ένα template, η `file` είναι η απλούστερη επιλογή. Το data source ταιριάζει σε αρχεία που φτιάχνει κάποιος άλλος ή κάποιο resource της ίδιας configuration.

Ρυθμίσεις από JSON

Ένα data source γίνεται χρήσιμο όταν οι τιμές του τροφοδοτούν resources. Εδώ μια άλλη ομάδα κρατά τις ρυθμίσεις μιας εφαρμογής σε αρχείο JSON. Εσύ θέλεις να φτιάξεις από αυτές ένα αρχείο `.env` χωρίς να αντιγράψεις τιμές με το χέρι.

TERMINAL

```
mkdir ../settings
cd ../settings
```

settings.json

```
{
  "app": "shop",
  "port": 8080,
  "features": ["cart", "search"]
}
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

data "local_file" "settings" {
  filename = "${path.module}/settings.json"
}

locals {
  settings = jsondecode(data.local_file.settings.content)
}

resource "local_file" "env" {
  filename = "${path.module}/out/app.env"
  content  = <<-EOT
    APP=${local.settings.app}
    PORT=${local.settings.port}
    FEATURES=${join(",", local.settings.features)}
  EOT
}
```

- Το data source διαβάζει το `settings.json` ως απλό κείμενο.
- Το local `settings` το περνά από τη `jsondecode` του κεφαλαίου 10 και το κάνει object. Μετά τα πεδία διαβάζονται με τελεία, όπως `local.settings.port`. Χωρίς το local θα έγραφες τη `jsondecode` τρεις φορές.

- Το resource `local_file.env` γράφει ένα heredoc με τρεις γραμμές. Η `join` ενώνει τη λίστα `features` με κόμματα.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.local_file.settings: Reading...
data.local_file.settings: Read complete after 0s [id=e55754bbe0eb5edc089d5544dd5b2b68f7dd0f17]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.env will be created
  + resource "local_file" "env" {
      + content              = <<-EOT
            APP=shop
            PORT=8080
            FEATURES=cart,search
        EOT
      …
      + filename             = "./out/app.env"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
local_file.env: Creating...
local_file.env: Creation complete after 0s [id=32ae1eb0bda3732e24923969c9d7d486fab2bb07]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το data source διαβάστηκε πρώτο, μέσα στο plan και πριν από τον υπολογισμό των αλλαγών. Έτσι το plan δείχνει ήδη το τελικό περιεχόμενο του `app.env`, με τις τιμές του JSON στη θέση τους.

TERMINAL

```
cat out/app.env
```

ΕΞΟΔΟΣ

```
APP=shop
PORT=8080
FEATURES=cart,search
```

Το state έχει τώρα και τα δύο είδη. Το πεδίο `mode` κάθε εγγραφής λέει ποιο είναι ποιο.

TERMINAL

```
jq '.resources[] | {mode, type, name}' terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "mode": "data",
  "type": "local_file",
  "name": "settings"
}
{
  "mode": "managed",
  "type": "local_file",
  "name": "env"
}
```

Το resource έχει `mode` ίσο με `managed` και το data source ίσο με `data`. Ο τύπος είναι ο ίδιος, `local_file`. Τη διαφορά την κάνει μόνο το `mode`, όπως στη διεύθυνση την κάνει το πρόθεμα `data`.

Η άλλη ομάδα αλλάζει τώρα τη θύρα. Εδώ το κάνει το `jq` του κεφαλαίου 4, που γράφει νέο αρχείο με το `port` στο `9090`.

TERMINAL

```
jq '.port = 9090' settings.json > settings.tmp
mv settings.tmp settings.json
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.env must be replaced
…
          - PORT=8080
          + PORT=9090
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Η νέα τιμή πέρασε από το data source στο local και από εκεί στο resource. Ο κώδικας δεν άλλαξε καθόλου. Το `app.env` θα αντικατασταθεί, γιατί κάθε αλλαγή στο `content` ενός `local_file` είναι replacement.

Τι κάνει το destroy

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
data.local_file.settings: Reading...
data.local_file.settings: Read complete after 0s [id=6a83e8ed2525a53b1ccd873170c41317101a69ae]
…
  # local_file.env will be destroyed
…
Plan: 0 to add, 0 to change, 1 to destroy.
local_file.env: Destroying... [id=32ae1eb0bda3732e24923969c9d7d486fab2bb07]
local_file.env: Destruction complete after 0s
…
Destroy complete! Resources: 1 destroyed.
```

Το destroy διάβασε κι αυτό το data source, όπως κάθε plan. Στο plan διαγραφής όμως υπάρχει μόνο το `local_file.env` και η περίληψη λέει `1 to destroy`.

TERMINAL

```
ls
terraform state list
```

ΕΞΟΔΟΣ

```
main.tf
out
settings.json
terraform.tfstate
terraform.tfstate.backup
```

Το `settings.json` έμεινε στη θέση του. Το `terraform state list` δεν τύπωσε τίποτα. Το data source έφυγε από το state μαζί με το resource, χωρίς να αγγίξει το αρχείο που διάβαζε. Ο φάκελος `out` έμεινε άδειος, γιατί ο provider σβήνει μόνο το αρχείο που έφτιαξε.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Το `terraform destroy` δεν σβήνει τίποτα για ένα data source. Σβήνει μόνο managed resources. Το αντικείμενο που διάβαζε το data source μένει όπως ήταν και η εγγραφή του φεύγει από το state.

Ένα zip με το archive_file

Στο κεφάλαιο 11 έφτιαξες zip με το resource `archive_file`. Ο provider `hashicorp/archive` δίνει τον ίδιο τύπο και ως data source, με τα ίδια arguments. Το zip το χρειάζεσαι όταν μια υπηρεσία θέλει τον κώδικα πακεταρισμένο. Το κλασικό παράδειγμα είναι μια function του AWS Lambda και εκεί θα βρεις σχεδόν πάντα τη μορφή του data source.

TERMINAL

```
mkdir -p ../zip/app
cd ../zip
```

app/handler.py

```
def handler(event, context):
    return {"message": "Γεια από τη Lambda"}
```

app/requirements.txt

```
requests==2.32.3
```

main.tf

```
terraform {
  required_providers {
    archive = {
      source  = "hashicorp/archive"
      version = "2.8.1"
    }
  }
}

data "archive_file" "app" {
  type        = "zip"
  source_dir  = "${path.module}/app"
  output_path = "${path.module}/build/app.zip"
}

output "size" {
  value = data.archive_file.app.output_size
}

output "hash" {
  value = data.archive_file.app.output_base64sha256
}
```

Τα `type` και `output_path` τα ξέρεις από το κεφάλαιο 11. Αντί για blocks `source` εδώ το `source_dir` βάζει στο zip ολόκληρο φάκελο. Με το `excludes`, μια λίστα ονομάτων σχετικά με το `source_dir`, αφήνεις έξω αρχεία του φακέλου. Για ένα μόνο αρχείο υπάρχει το `source_file`.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
data.archive_file.app: Reading...
data.archive_file.app: Read complete after 0s [id=94e79e7ce9deb511ee7b1a5e85955bcada4e042a]

Changes to Outputs:
  + hash = "Q1f+FzkfeWEAUDrG+lw5D3QVekfE+9UYfukQMMj1tWM="
  + size = 372

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το `size` είναι το μέγεθος του zip σε bytes. Το `hash` είναι το `output_base64sha256`, δηλαδή το SHA256 του zip γραμμένο σε base64. Ένα AWS Lambda θέλει αυτή ακριβώς τη μορφή στο argument `source_code_hash`. Όταν αλλάζει ο κώδικας, αλλάζει το hash και το Terraform ξέρει ότι πρέπει να ανεβάσει νέα έκδοση.

TERMINAL

```
ls build
```

ΕΞΟΔΟΣ

```
app.zip
```

Το zip υπάρχει ήδη, αν και έτρεξε μόνο plan. Το data source γράφει το αρχείο του τη στιγμή που διαβάζεται και το plan διαβάζει τα data sources. Το resource του κεφαλαίου 11 γράφει το zip μόνο στο apply.

ΠΑΓΙΔΑ

Το data source `archive_file` είναι εξαίρεση στον κανόνα ότι ένα data source δεν αλλάζει τίποτα. Γράφει το zip στον δίσκο σε κάθε ανάγνωση, ακόμη και στο plan. Το destroy δεν το σβήνει. Το zip δεν είναι υποδομή. Είναι ένα τοπικό αρχείο που ξαναγράφεται σε κάθε plan.

Κάνε apply για να μείνουν οι τιμές στο state. Μετά άλλαξε μία γραμμή του κώδικα.

TERMINAL

```
terraform apply -auto-approve
```

app/handler.py

```
def handler(event, context):
    return {"message": "Γεια από τη Lambda, έκδοση 2"}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  ~ hash = "Q1f+FzkfeWEAUDrG+lw5D3QVekfE+9UYfukQMMj1tWM=" -> "vncoKp3Q40Uho9hV2O8EbX2y8EdKULCNpKY1N9j07gU="
  ~ size = 372 -> 390
…
```

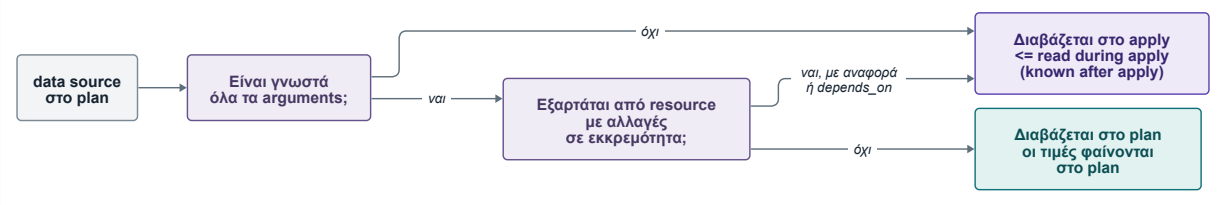
Το hash άλλαξε μαζί με τον κώδικα και το μέγεθος μεγάλωσε. Ο provider γράφει σταθερή ημερομηνία σε κάθε αρχείο του zip, οπότε το hash αλλάζει μόνο όταν αλλάξει το περιεχόμενο. Ένα αρχείο που απλώς αποθηκεύτηκε ξανά δεν το επηρεάζει. Τα δικαιώματα των αρχείων όμως μετρούν. Δύο μηχανήματα με διαφορετικά δικαιώματα στα ίδια αρχεία βγάζουν διαφορετικό hash. Το `output_file_mode = "0644"` δίνει σε όλα τα αρχεία του zip τα ίδια δικαιώματα και κάνει το hash ίδιο παντού.



Πότε διαβάζεται ένα data source

Σε όλα τα ως τώρα παραδείγματα το data source διαβάστηκε μέσα στο plan. Αυτή είναι η προεπιλογή, γιατί έτσι οι τιμές του φαίνονται στο plan και τις ελέγχεις πριν από το apply. Η ανάγνωση μετακινείται στο apply σε δύο περιπτώσεις.

1. Κάποιο argument έχει τιμή που θα είναι γνωστή μόνο μετά το apply, δηλαδή `(known after apply)`.
2. Το data source εξαρτάται από resource που έχει αλλαγές σε εκκρεμότητα στο ίδιο plan, με `depends_on` ή με αναφορά γραμμένη μέσα στο ίδιο το block `data`.



Σχήμα 12.1 · Πότε διαβάζεται ένα data source. Αν όλα τα arguments είναι γνωστά και δεν περιμένει κανένα resource με αλλαγές, διαβάζεται στο plan. Αλλιώς η ανάγνωση γίνεται στο apply και το plan γράφει (known after apply) στα attributes του.

Το επόμενο παράδειγμα δείχνει την πρώτη περίπτωση. Κάθε release παίρνει ένα τυχαίο όνομα από το `random_pet` του κεφαλαίου 6. Ένα `local_file` γράφει τις σημειώσεις του release σε αρχείο με αυτό το όνομα και το `archive_file` το βάζει σε zip.

TERMINAL

```
mkdir ../release
cd ../release
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
```

```
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
    archive = {
      source  = "hashicorp/archive"
      version = "2.8.1"
    }
  }
}

resource "random_pet" "release" {}

resource "local_file" "notes" {
  filename = "${path.module}/notes/${random_pet.release.id}.txt"
  content  = "Release ${random_pet.release.id}\n"
}

data "archive_file" "release" {
  type        = "zip"
  source_file = local_file.notes.filename
  output_path = "${path.module}/dist/release.zip"
}

output "zip_size" {
  value = data.archive_file.release.output_size
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create
 <= read (data resources)

Terraform will perform the following actions:

  # data.archive_file.release will be read during apply
  # (config refers to values not yet known)
 <= data "archive_file" "release" {
      + id                  = (known after apply)
      + output_base64sha256 = (known after apply)
      …
      + output_path         = "./dist/release.zip"
      …
      + output_size         = (known after apply)
      + source_file         = (known after apply)
      + type                = "zip"
    }

  # local_file.notes will be created
```

```
  + resource "local_file" "notes" {
      + content              = (known after apply)
      …
      + filename             = (known after apply)
      + id                   = (known after apply)
    }

  # random_pet.release will be created
  + resource "random_pet" "release" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + zip_size = (known after apply)

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Στη λίστα των συμβόλων στην κορυφή εμφανίστηκε το `<=` με την εξήγηση `read (data resources)`. Είναι το σύμβολο της ανάγνωσης και ένα data source το παίρνει στο plan μόνο όταν η ανάγνωση γίνεται στο apply. Το σχόλιο από πάνω λέει ρητά `will be read during apply`. Η επόμενη γραμμή λέει την αιτία, `config refers to values not yet known`.

Το `source_file` είναι `(known after apply)`, γιατί το όνομα του αρχείου περιέχει το τυχαίο όνομα, που δεν υπάρχει ακόμη. Τα attributes που υπολογίζει το data source, όπως το `output_size`, είναι κι αυτά άγνωστα. Γι' αυτό και το output `zip_size` γράφει `(known after apply)`. Η περίληψη μετρά μόνο τα δύο resources.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.release: Creating...
random_pet.release: Creation complete after 0s [id=whole-eel]
local_file.notes: Creating...
local_file.notes: Creation complete after 0s [id=6ca05b05245f25af60bb88e3050cf95a00252a1e]
data.archive_file.release: Reading...
data.archive_file.release: Read complete after 0s [id=5e4d4164d997ed56742eb0351af33d0972e317ab]
…
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
…
zip_size = 164
```

Η σειρά ακολουθεί τις εξαρτήσεις. Πρώτα το `random_pet`, μετά το `local_file` και τελευταία η ανάγνωση του data source, που βρήκε πλέον το αρχείο. Το μέγεθος του zip εξαρτάται από το τυχαίο όνομα, οπότε το δικό σου θα διαφέρει λίγο.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.release: Refreshing state... [id=whole-eel]
local_file.notes: Refreshing state... [id=6ca05b05245f25af60bb88e3050cf95a00252a1e]
data.archive_file.release: Reading...
data.archive_file.release: Read complete after 0s [id=5e4d4164d997ed56742eb0351af33d0972e317ab]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Μετά το apply όλα τα arguments είναι γνωστά και κανένα resource δεν έχει αλλαγές. Το data source διαβάζεται πάλι μέσα στο plan, όπως στα πρώτα παραδείγματα. Το plan λέει `No changes`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Στο plan το σύμβολο `<=` σημαίνει ανάγνωση data source κατά το apply. Ένα data source διαβάζεται στο plan όταν όλα τα arguments του είναι γνωστά. Αν κάποιο εξαρτάται από τιμή που θα υπάρξει μόνο μετά το apply, η ανάγνωση γίνεται στο apply και τα attributes του είναι `(known after apply)` στο plan.

depends_on σε data source

Η δεύτερη περίπτωση είναι η εξάρτηση από resource με αλλαγές. Το `depends_on` του κεφαλαίου 6 γράφεται και σε block `data`, αλλά εκεί έχει μια παρενέργεια που πρέπει να ξέρεις. Στο παράδειγμα ένα data source διαβάζει τον αριθμό έκδοσης από το `version.txt`. Δεν έχει καμία σχέση με το `motd.txt`, όμως κάποιος έβαλε `depends_on` για σιγουριά. Το data source μπαίνει σε δικό του αρχείο, το `version.tf`, ώστε να το αλλάξεις μετά χωρίς να ξαναγράψεις όλο το `main.tf`.

TERMINAL

```
mkdir ../pinned
cd ../pinned
echo "v1" > version.txt
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "message" {
  type    = string
  default = "Καλημέρα"
}

resource "local_file" "motd" {
```

```
  filename = "${path.module}/motd.txt"
  content  = "${var.message}\n"
}

resource "local_file" "stamp" {
  filename = "${path.module}/stamp.txt"
  content  = "Έκδοση ${data.local_file.version.content}"
}
```

version.tf

```
data "local_file" "version" {
  filename = "${path.module}/version.txt"

  depends_on = [local_file.motd]
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

Άλλαξε τώρα μόνο το μήνυμα με `-var`, όπως στο κεφάλαιο 7.

TERMINAL

```
terraform plan -var message=Καληνύχτα
```

ΕΞΟΔΟΣ

```
…
  # data.local_file.version will be read during apply
  # (depends on a resource or a module with changes pending)
…
  # local_file.motd must be replaced
…
      ~ content              = <<-EOT # forces replacement
…
  # local_file.stamp must be replaced
…
        EOT -> (known after apply) # forces replacement
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Το `motd` αντικαθίσταται, όπως περίμενες. Το data source όμως εξαρτάται από αυτό με `depends_on`, οπότε η ανάγνωση μετακινήθηκε στο apply με αιτία `depends on a resource or a module with changes pending`. Το `content` του data source έγινε άγνωστο. Το `stamp` το χρησιμοποιεί, οπότε το plan θέλει να αντικαταστήσει κι αυτό. Το περιεχόμενό του θα βγει ίδιο, `Έκδοση v1`, αλλά το Terraform δεν μπορεί να το ξέρει στο plan. Ένα περιττό `depends_on` έκανε μια αλλαγή σε δύο replacements.

Σβήσε το `depends_on` και τρέξε ξανά το ίδιο plan.

version.tf

```
data "local_file" "version" {
  filename = "${path.module}/version.txt"
}
```

TERMINAL

```
terraform plan -var message=Καληνύχτα
```

ΕΞΟΔΟΣ

```
…
  # local_file.motd must be replaced
…
      ~ content              = <<-EOT # forces replacement
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Χωρίς το `depends_on` το data source διαβάζεται στο plan, το `stamp` μένει ίδιο και αντικαθίσταται μόνο το `motd`.

> **ΠΑΓΙΔΑ**
> Μη βάζεις `depends_on` σε data source χωρίς λόγο. Κάθε φορά που η εξάρτηση έχει αλλαγές, η ανάγνωση πηγαίνει στο apply. Ό,τι χρησιμοποιεί το data source γίνεται `(known after apply)` και μπορεί να φέρει replacements που δεν χρειάζονται. Αν το data source διαβάζει κάτι που φτιάχνει ένα resource, γράψε αναφορά σε attribute του resource, όπως το `local_file.notes.filename` στο `release`. Η αναφορά φέρνει την ίδια εξάρτηση και δείχνει στον κώδικα γιατί υπάρχει. Το `depends_on` μένει για εξαρτήσεις που δεν περνούν από καμία τιμή.

count και for_each σε data source

Τα meta-arguments `count` και `for_each` του κεφαλαίου 11 δουλεύουν και στα blocks `data`. Ένα block διαβάζει τότε πολλά παρόμοια αντικείμενα. Εδώ κάθε μέλος μιας ομάδας έχει δικό του αρχείο.

TERMINAL

```
mkdir -p ../team/members
cd ../team
echo "Άννα, backend" > members/anna.txt
echo "Νίκος, frontend" > members/nikos.txt
echo "Ελένη, δίκτυα" > members/eleni.txt
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

```
variable "members" {
  type    = set(string)
  default = ["anna", "nikos", "eleni"]
}

data "local_file" "member" {
  for_each = var.members
  filename = "${path.module}/members/${each.key}.txt"
}

resource "local_file" "roster" {
  filename = "${path.module}/out/roster.txt"
  content  = join("", [for m in data.local_file.member : m.content])
}
```

Το `for_each` φτιάχνει ένα instance για κάθε στοιχείο του set και το `each.key` δίνει το όνομα του αρχείου. Το `data.local_file.member` είναι πλέον map με κλειδί το όνομα του μέλους. Το `for` του κεφαλαίου 10 περνά από τις τιμές του και η `join` ενώνει τα περιεχόμενα.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
data.local_file.member["anna"]: Read complete after 0s [id=3d0b708d3ffc4f5d28b99b496ec6f00bf8e2d62d]
data.local_file.member["nikos"]: Read complete after 0s [id=653f54b024373803e018343d3baeede5ad2da369]
data.local_file.member["eleni"]: Read complete after 0s [id=3b48e780948d2f05b35965b199188ecdc03ddee6]
…
local_file.roster: Creating...
local_file.roster: Creation complete after 0s [id=72ae82cc63041d4935198d6508a862c579c2883b]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Οι τρεις αναγνώσεις είναι ανεξάρτητες και γίνονται παράλληλα, οπότε η σειρά τους στο δικό σου terminal μπορεί να διαφέρει.

TERMINAL

```
terraform state list
cat out/roster.txt
```



```
ΕΞΟΔΟΣ

data.local_file.member["anna"]
data.local_file.member["eleni"]
data.local_file.member["nikos"]
local_file.roster
Άννα, backend
Ελένη, δίκτυα
Νίκος, frontend
```

Κάθε instance έχει δική του διεύθυνση, με το κλειδί σε αγκύλες, όπως `data.local_file.member["anna"]`. Με `count` θα ήταν `data.local_file.member[0]`. Το `roster.txt` έχει τα μέλη με τη σειρά των κλειδιών, γιατί το `for` διαβάζει ένα map με ταξινομημένα κλειδιά.

Data sources στο cloud

Τα data sources αυτού του κεφαλαίου διαβάζουν τοπικά αρχεία, για να βλέπεις τον μηχανισμό χωρίς λογαριασμό. Η πραγματική τους αξία φαίνεται στους providers του cloud. Τα παρακάτω αποσπάσματα δεν εκτελούνται εδώ. Θα τα γράψεις στα κεφάλαια 17 και 18.

```hcl
data "aws_caller_identity" "current" {}

data "aws_availability_zones" "available" {
  state = "available"
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd-gp3/ubuntu-noble-24.04-amd64-server-*"]
  }
}

resource "aws_instance" "web" {
```

```
  ami               = data.aws_ami.ubuntu.id
  instance_type     = "t3.micro"
  availability_zone = data.aws_availability_zones.available.names[0]
}
```

- Το `aws_caller_identity` δεν παίρνει arguments. Επιστρέφει τον λογαριασμό στον οποίο ανήκουν τα credentials σου, για παράδειγμα στο `account_id`.
- Το `aws_availability_zones` επιστρέφει στο `names` τις availability zones του region.
- Το `aws_ami` ψάχνει image. Το `owners` περιορίζει την αναζήτηση στον λογαριασμό της Canonical, που δημοσιεύει τα images του Ubuntu. Το block `filter` ταιριάζει το όνομα με μοτίβο και το `most_recent` κρατά το νεότερο. Έτσι ο server παίρνει το πιο πρόσφατο image χωρίς ID γραμμένο με το χέρι, που θα ήταν διαφορετικό σε κάθε region.

Ένα ακόμη data source έρχεται ενσωματωμένο στο Terraform, όπως το `terraform_data` του κεφαλαίου 6. Το `terraform_remote_state` διαβάζει τα outputs από το state μιας άλλης configuration. Έτσι η configuration του δικτύου και η configuration των servers μένουν χωριστές, αλλά η δεύτερη βρίσκει το subnet της πρώτης. Το κεφάλαιο 19 το δείχνει με S3 backend.

HCL

```
data "terraform_remote_state" "network" {
  backend = "s3"

  config = {
    bucket = "tfbook-state"
    key    = "network/terraform.tfstate"
    region = "eu-central-1"
  }
}

resource "aws_instance" "app" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
  subnet_id     = data.terraform_remote_state.network.outputs.subnet_id
}
```

Resource και data σε μία σελίδα

	`resource`	`data`
Τι κάνει	Φτιάχνει, αλλάζει, σβήνει	Μόνο διαβάζει
Διεύθυνση	`local_file.env`	`data.local_file.settings`
`mode` στο state	`managed`	`data`
Πότε δουλεύει	Στο apply	Στο plan, ή στο apply αν περιμένει κάτι
Σύμβολο στο plan	`+`, `~`, `-`, `-/+`	`<=` όταν διαβάζεται στο apply
Στην περίληψη `Plan:`	Μετρά	Δεν μετρά
Στο destroy	Σβήνεται	Φεύγει από το state, το αντικείμενο μένει

	resource	data
lifecycle	Όλα τα arguments	Μόνο precondition και postcondition

Τα precondition και postcondition είναι το θέμα του κεφαλαίου 13. Ένα create_before_destroy ή prevent_destroy σε block data είναι σφάλμα, γιατί ένα data source δεν έχει τίποτα να αντικαταστήσει ή να σβήσει.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο ~/tfbook/ch12. Τα αρχεία που διαβάζουν τα data sources τα φτιάχνεις με echo ή όπως λέει η εκφώνηση. Κάθε configuration δηλώνει τους providers που χρησιμοποιεί, με τις εκδόσεις του κεφαλαίου.

ΑΣΚΗΣΗ 12.1 Ένα αρχείο από έξω

Φτιάξε με echo το αρχείο owner.txt με περιεχόμενο Μαρία. Γράψε configuration με ένα data source local_file που το διαβάζει και ένα output owner με το περιεχόμενο χωρίς την αλλαγή γραμμής. Κάνε apply και τύπωσε το output με terraform output -raw owner.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Μαρία
```

ΑΣΚΗΣΗ 12.2 Managed και data

Φτιάξε με echo το motto.txt με περιεχόμενο less is more. Γράψε configuration με ένα data source motto που το διαβάζει και ένα resource local_file με όνομα loud, που γράφει το ίδιο κείμενο με κεφαλαία στο out/motto.txt. Μετά το apply τύπωσε με jq -c έναν πίνακα [mode, type, name] για κάθε εγγραφή του state. Το -c γράφει κάθε αποτέλεσμα σε μία γραμμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
["data","local_file","motto"]
["managed","local_file","loud"]
```

ΑΣΚΗΣΗ 12.3 Θύρες από JSON

Γράψε στον φάκελο της άσκησης το παρακάτω `ports.json`. Διάβασέ το με data source και γράψε output `high` με τα ονόματα των services που ακούνε σε θύρα πάνω από `8500`. Κάνε apply και τύπωσε το output με `terraform output -json high`.

ports.json

```
{
  "web": 8080,
  "api": 9090,
  "admin": 9000,
  "metrics": 2112
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
["admin","api"]
```

ΑΣΚΗΣΗ 12.4 Plan ή apply;

Ξεκίνα με το παρακάτω `main.tf`. Πριν τρέξεις οτιδήποτε, σκέψου αν το data source θα διαβαστεί στο plan ή στο apply. Τρέξε `terraform plan` και βρες τις γραμμές που το επιβεβαιώνουν. Μετά κάνε apply και τρέξε ξανά plan.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_integer" "build" {
  min = 1
  max = 999
}

resource "local_file" "build" {
  filename = "${path.module}/builds/${random_integer.build.result}.txt"
  content  = "build ${random_integer.build.result}\n"
}

data "local_file" "build" {
  filename = local_file.build.filename
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # data.local_file.build will be read during apply
  # (config refers to values not yet known)
…
Plan: 2 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 12.5 Zip χωρίς σημειώσεις

Φτιάξε τον φάκελο `web` με τρία αρχεία, τα `index.html`, `style.css` και `notes.md`, με όποιο περιεχόμενο θέλεις. Γράψε ένα data source `archive_file` που φτιάχνει το `build/site.zip` από τον φάκελο χωρίς το `notes.md`. Τρέξε μόνο `terraform plan` και δείξε τα αρχεία του zip με `unzip -Z1 build/site.zip`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
index.html
style.css
```

ΑΣΚΗΣΗ 12.6 Ένα αρχείο ανά περιβάλλον

Φτιάξε τα αρχεία `env/dev.txt`, `env/stage.txt` και `env/prod.txt` με τους αριθμούς `1`, `2` και `3`. Είναι τα replicas κάθε περιβάλλοντος. Με ένα μόνο block `data` και `for_each` διάβασε και τα τρία. Γράψε output `replicas`, ένα map από περιβάλλον σε αριθμό. Κάνε apply και τύπωσε το output.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
{
  "dev" = 1
  "prod" = 3
  "stage" = 2
}
```

ΑΣΚΗΣΗ 12.7 Το αρχείο που δεν υπήρχε ακόμη

Ξεκίνα με το παρακάτω `main.tf`. Τρέξε `terraform apply -auto-approve` και διάβασε το σφάλμα. Διόρθωσε το data source ώστε ένα apply σε καθαρό φάκελο να πετυχαίνει με την πρώτη. Τύπωσε μετά το output με `terraform output -raw config`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "config" {
  filename = "${path.module}/out/config.txt"
  content  = "mode=fast\n"
}

data "local_file" "config" {
  filename = "${path.module}/out/config.txt"
}

output "config" {
  value = trimspace(data.local_file.config.content)
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
mode=fast
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 12.1

Ποια πρόταση περιγράφει σωστά τη διαφορά ανάμεσα σε ένα block `resource` και ένα block `data`;

a. Το `data` φτιάχνει το αντικείμενο μόνο αν δεν υπάρχει ήδη, ενώ το `resource` το φτιάχνει πάντα.
b. Το `resource` διαχειρίζεται τον κύκλο ζωής ενός αντικειμένου, ενώ το `data` μόνο διαβάζει πληροφορία για κάτι που υπάρχει ήδη.
c. Το `data` δεν γράφεται ποτέ στο state, ενώ το `resource` γράφεται πάντα.
d. Το `data` υπάρχει μόνο σε providers που δεν μιλούν με cloud, όπως ο `local`.

ΕΡΩΤΗΣΗ 12.2

Μια configuration περιέχει το block `data "aws_ami" "web"`. Ποια έκφραση δίνει το ID του image σε ένα resource;

a. `aws_ami.web.id`
b. `data.web.id`
c. `data.aws_ami.web.id`
d. `ami.data.web.id`

ΕΡΩΤΗΣΗ 12.3

Το plan δείχνει `<= data "local_file" "report"` με το σχόλιο `will be read during apply`. Ποια από τις παρακάτω μπορεί να είναι η αιτία;

a. Κάποιο argument του data source εξαρτάται από attribute ενός resource που θα δημιουργηθεί στο ίδιο apply.
b. Το αρχείο που διαβάζει το data source δεν υπάρχει.
c. Το plan τρέχει χωρίς `-out`, οπότε τα data sources διαβάζονται μόνο στο apply.
d. Τα data sources διαβάζονται πάντα στο apply και το σχόλιο εμφανίζεται σε κάθε plan.

ΕΡΩΤΗΣΗ 12.4

Μια configuration έχει ένα resource `local_file` και ένα data source `local_file` που διαβάζει ένα αρχείο άλλης ομάδας. Τι κάνει το `terraform destroy`;

a. Σβήνει και τα δύο αρχεία, γιατί και τα δύο είναι γραμμένα στο state.
b. Σβήνει μόνο το αρχείο του resource. Το αρχείο της άλλης ομάδας μένει και το data source φεύγει από το state.
c. Αρνείται να τρέξει, γιατί ένα data source δεν μπορεί να σβηστεί.
d. Σβήνει μόνο το αρχείο του data source, γιατί διαβάστηκε τελευταίο.

ΕΡΩΤΗΣΗ 12.5

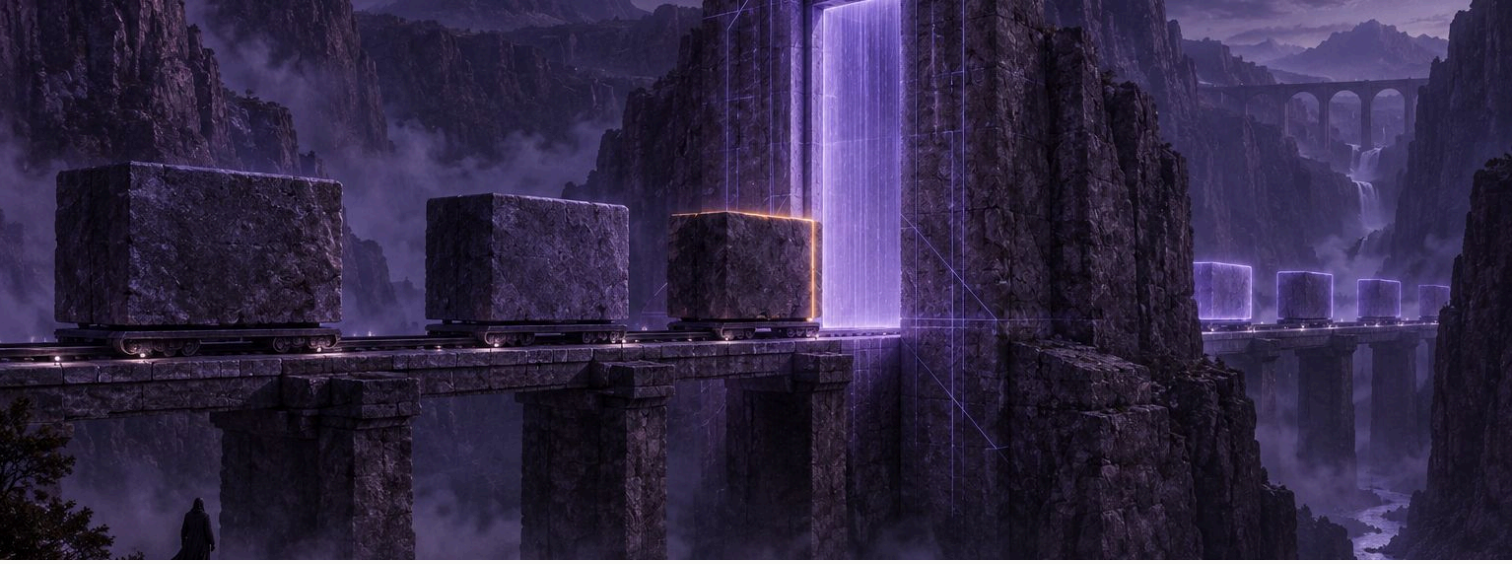
Ποια από τα παρακάτω επιτρέπονται σε ένα block `data`; Επίλεξε δύο.

a. `for_each`
b. `lifecycle { prevent_destroy = true }`
c. `depends_on`
d. `lifecycle { create_before_destroy = true }`

ΕΡΩΤΗΣΗ 12.6

Σωστό ή λάθος; Το `terraform validate` αποτυγχάνει όταν λείπει το αρχείο που διαβάζει ένα data source `local_file`.

a. Σωστό
b. Λάθος



ΚΕΦΑΛΑΙΟ 13

Validation, preconditions και check blocks

Κάθε configuration στηρίζεται σε υποθέσεις. Ότι το περιβάλλον λέγεται `prod` και όχι `prodution`, ότι ένα αρχείο που διαβάζεις έχει τα πεδία που περιμένεις, ότι ένα resource βγήκε όπως το ήθελες. Σε αυτό το κεφάλαιο γράφεις αυτές τις υποθέσεις ως ελέγχους. Γνωρίζεις τους τέσσερις μηχανισμούς του Terraform, πότε τρέχει ο καθένας, ποιοι σταματούν τη δουλειά και ποιος μόνο προειδοποιεί. Μαθαίνεις επίσης να γράφεις μηνύματα που λένε στον χρήστη τι να διορθώσει.

Υποθέσεις που γίνονται έλεγχοι

Ο τύπος μιας μεταβλητής, από το κεφάλαιο 7, πιάνει ένα είδος λάθους. Ένα `string` δεν δέχεται λίστα και ένα `number` δεν δέχεται το `πέντε`. Οι περισσότερες λάθος τιμές όμως έχουν σωστό τύπο. Το `prodution` είναι κανονικό string και το `-3` κανονικός αριθμός. Το Terraform θα τα δεχτεί. Θα φτιάξει ένα αρχείο με λάθος όνομα ή θα στείλει την τιμή στον provider, που θα απαντήσει με δικό του σφάλμα, μακριά από την αιτία.

Χρειάζεσαι έλεγχο που λέει «αυτή η τιμή έχει νόημα για την υποδομή μου». Το Terraform σου δίνει τέσσερις τρόπους να τον γράψεις.

1. Το block `validation` μέσα σε μια μεταβλητή ελέγχει την τιμή που δίνει ο χρήστης.
2. Το `precondition` ελέγχει μια υπόθεση πριν φτιαχτεί ή αλλάξει ένα resource, πριν διαβαστεί ένα data source ή πριν γραφτεί ένα output.

3. Το `postcondition` ελέγχει το αποτέλεσμα, αφού φτιαχτεί, αλλάξει ή διαβαστεί το αντικείμενο.
4. Το block `check` ελέγχει την υποδομή στο σύνολό της και μόνο προειδοποιεί.

Όλοι έχουν το ίδιο σχήμα. Ένα `condition`, δηλαδή μια έκφραση που βγάζει `true` ή `false`. Ένα `error_message`, δηλαδή το κείμενο που βλέπει ο χρήστης όταν το `condition` βγει `false`. Διαφέρουν στο πού γράφονται, στο πότε ελέγχονται και στο τι κάνει το Terraform όταν αποτύχουν.

Validation σε μεταβλητή

Φτιάξε φάκελο για το πρώτο παράδειγμα. Μια μικρή εφαρμογή γράφει το αρχείο ρυθμίσεών της, με όνομα που βγαίνει από το περιβάλλον.

TERMINAL

```
mkdir app
cd app
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

variables.tf

```
variable "env" {
  type        = string
  description = "Το περιβάλλον της εφαρμογής"
  default     = "dev"

  validation {
    condition     = contains(["dev", "staging", "prod"], var.env)
    error_message = "Το env πρέπει να είναι dev, staging ή prod."
  }
}
```

main.tf

```
resource "local_file" "conf" {
  filename = "${var.env}.conf"
  content  = "env = ${var.env}\n"
}
```

Το block `validation` μπαίνει μέσα στο block `variable`. Το `condition` πρέπει να βγάλει `true` για να γίνει δεκτή η τιμή. Εδώ η `contains` του κεφαλαίου 10 ελέγχει αν το `var.env` υπάρχει στη λίστα με τα τρία επιτρεπτά ονόματα. Το `error_message` είναι το μήνυμα για όποιον δώσει άλλη τιμή.

Κάνε init και δώσε στο plan μια τιμή με ορθογραφικό λάθος.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan -var env=prodution
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Invalid value for variable
│
│   on variables.tf line 1:
│    1: variable "env" {
│      ├────────────────
│      │ var.env is "prodution"
│
│ Το env πρέπει να είναι dev, staging ή prod.
│
│ This was checked by the validation rule at variables.tf:6,3-13.
╵
```

Το plan σταμάτησε με `Planning failed`. Το σφάλμα `Invalid value for variable` δείχνει τη δήλωση της μεταβλητής και την τιμή που ήρθε, `var.env is "prodution"`. Από κάτω βρίσκεται το δικό σου μήνυμα. Η τελευταία γραμμή λέει ποιος κανόνας έπιασε την τιμή. Είναι το block `validation` στη γραμμή 6 του `variables.tf`, από τη στήλη 3 έως τη 13.

Πότε ελέγχεται

Η validation τρέχει στο plan, μόλις το Terraform μάθει την τιμή της μεταβλητής. Αυτό γίνεται πριν από το refresh και τον σχεδιασμό κάθε resource που χρησιμοποιεί τη μεταβλητή. Γι' αυτό η έξοδος δεν έχει ούτε μία γραμμή για το `local_file.conf`. Το apply ξεκινά πάντα με plan, οπότε ελέγχει με τον ίδιο τρόπο πριν ρωτήσει οτιδήποτε. Μια τιμή που δεν περνά δεν φτάνει ποτέ σε resource ή σε provider.

Το `terraform validate` του κεφαλαίου 2 ελέγχει την configuration χωρίς τις τιμές των μεταβλητών. Δεν χρησιμοποιεί ούτε το `default`. Ελέγχει λοιπόν ότι ο κανόνας είναι σωστά γραμμένος, όχι αν μια συγκεκριμένη τιμή τον περνά. Ένα `default` που παραβιάζει τον δικό του κανόνα περνά το `validate` και πιάνεται στο πρώτο plan. Εξαίρεση είναι μια τιμή γραμμένη απευθείας στον κώδικα, όπως ένα argument σε block `module` του κεφαλαίου 15. Αυτή είναι γνωστή από πριν και το `validate` την ελέγχει.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Η validation ελέγχει την τιμή μιας μεταβλητής στο plan, πριν τη χρησιμοποιήσει οποιοδήποτε resource. Όταν αποτύχει, βγάζει error και το plan σταματά χωρίς να αλλάξει τίποτα.

Συναρτήσεις για conditions

Τα περισσότερα conditions χτίζονται με λίγες συναρτήσεις. Η `contains` ελέγχει λίστα επιτρεπτών τιμών και η `length` μετρά χαρακτήρες ή στοιχεία. Για τη μορφή ενός string υπάρχει η `regex`, που δεν την έχεις δει ακόμη. Δοκίμασέ τες στο console.

TERRAFORM CONSOLE

```
> regex("^[a-z][a-z0-9-]*$", "web-01")
"web-01"
> regex("^[a-z][a-z0-9-]*$", "Web_01")
│ Error: Error in function call
│
│   on <console-input> line 1:
│   (source code not available)
│
│ Call to function "regex" failed: pattern did not match any part of the
│ given string.
> can(regex("^[a-z][a-z0-9-]*$", "Web_01"))
false
> length("web-01")
6
> contains(["dev", "staging", "prod"], "qa")
false
```

Η `regex` δέχεται ένα pattern και ένα string και επιστρέφει το κομμάτι που ταίριαξε. Στο pattern το `^` σημαίνει αρχή και το `$` τέλος του string. Το `[a-z]` είναι ένα πεζό λατινικό γράμμα και το `[a-z0-9-]*` οσαδήποτε πεζά, ψηφία ή παύλες. Το `web-01` ταιριάζει ολόκληρο, οπότε επιστρέφεται ολόκληρο.

Όταν το string δεν ταιριάζει, η `regex` δεν δίνει `false`. Δίνει σφάλμα, `Call to function "regex" failed`. Ένα condition όμως πρέπει να βγάλει bool. Γι' αυτό τυλίγεις την `regex` σε `can`, που όπως είδες στο κεφάλαιο 10 κάνει το σφάλμα `false`. Το `can(regex(...))` είναι ο συνηθισμένος τρόπος να ελέγξεις τη μορφή ενός string.

> **ΠΑΓΙΔΑ**
>
> Αν γράψεις `regex(...)` χωρίς `can` μέσα σε condition, ο χρήστης δεν βλέπει ποτέ το `error_message` σου όταν η τιμή δεν ταιριάζει. Βλέπει μόνο το σφάλμα της συνάρτησης, `Error in function call`. Πρέπει να μαντέψει τι ήθελες.

Πολλά validation blocks

Μια μεταβλητή μπορεί να έχει όσα blocks `validation` θέλεις. Πρόσθεσε στο `variables.tf` το όνομα της υπηρεσίας με δύο κανόνες.

variables.tf, στο τέλος

```
variable "service" {
  type        = string
  description = "Το όνομα της υπηρεσίας, μπαίνει στο όνομα του αρχείου"
  default     = "web"

  validation {
    condition     = can(regex("^[a-z][a-z0-9-]*$", var.service))
    error_message = "Το service ξεκινά με πεζό γράμμα και έχει μόνο a έως z, ψηφία και παύλες."
  }

  validation {
    condition     = length(var.service) <= 12
    error_message = "Το service έχει έως 12 χαρακτήρες. Το ${var.service} έχει ${length(var.service)}."
  }
}
```

main.tf

```
resource "local_file" "conf" {
  filename = "${var.service}-${var.env}.conf"
  content  = "service = ${var.service}\nenv = ${var.env}\n"
}
```

TERMINAL

```
terraform plan -var service=Web_Server_01
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Invalid value for variable
│
│   on variables.tf line 11:
│   11: variable "service" {
│     ├────────────────
│     │ var.service is "Web_Server_01"
│
│ Το service ξεκινά με πεζό γράμμα και έχει μόνο a έως z, ψηφία και παύλες.
│
│ This was checked by the validation rule at variables.tf:16,3-13.
╵

╷
│ Error: Invalid value for variable
│
│   on variables.tf line 11:
│   11: variable "service" {
│     ├────────────────
│     │ var.service is "Web_Server_01"
│
│ Το service έχει έως 12 χαρακτήρες. Το Web_Server_01 έχει 13.
│
│ This was checked by the validation rule at variables.tf:21,3-13.
╵
```

Η τιμή παραβιάζει και τους δύο κανόνες, οπότε βλέπεις δύο σφάλματα, ένα για κάθε block. Το Terraform δεν σταματά στον πρώτο κανόνα που αποτυγχάνει. Τους ελέγχει όλους και τους αναφέρει μαζί, ώστε να διορθώσεις τα πάντα με μία προσπάθεια. Γι' αυτό γράφεις έναν κανόνα για κάθε απαίτηση, με δικό του μήνυμα. Ένα μεγάλο condition με `&&` θα έδινε ένα γενικό μήνυμα για όλα.

Το δεύτερο μήνυμα έχει μέσα του εκφράσεις. Το `error_message` είναι string template, όπως όσα είδες στο κεφάλαιο 10. Έτσι το μήνυμα λέει `Το Web_Server_01 έχει 13`, δηλαδή ακριβώς πόσο απέχει η τιμή από το όριο.

Validation που κοιτάζει άλλη μεταβλητή

Κάποιοι κανόνες αφορούν δύο μεταβλητές μαζί. Στο production θέλεις τουλάχιστον δύο αντίγραφα της υπηρεσίας, ενώ στο dev αρκεί ένα. Πρόσθεσε στο τέλος του `variables.tf` τη μεταβλητή `replicas` και γράψε ξανά το `main.tf`.

variables.tf, στο τέλος

```
variable "replicas" {
  type        = number
  description = "Πόσα αντίγραφα της υπηρεσίας τρέχουν"
  default     = 1

  validation {
    condition     = var.replicas >= 1 && var.replicas <= 10
    error_message = "Το replicas πρέπει να είναι από 1 έως 10."
  }

  validation {
    condition     = var.env != "prod" || var.replicas >= 2
    error_message = "Το prod θέλει τουλάχιστον 2 replicas, ώστε να αντέχει την πτώση ενός."
  }
}
```

main.tf

```
resource "local_file" "conf" {
  filename = "${var.service}-${var.env}.conf"
  content  = "service = ${var.service}\nenv = ${var.env}\nreplicas = ${var.replicas}\n"
}
```

Το δεύτερο condition διαβάζεται «ή το env δεν είναι prod ή τα replicas είναι τουλάχιστον 2». Όταν το env δεν είναι `prod`, το πρώτο σκέλος είναι `true` και ο κανόνας περνά. Όταν είναι `prod`, μετρά μόνο το δεύτερο. Έτσι γράφεται κάθε κανόνας της μορφής «αν ισχύει το A, πρέπει να ισχύει και το B».

TERMINAL

```
terraform plan -var env=prod
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Invalid value for variable
│
│   on variables.tf line 26:
│   26: variable "replicas" {
│     ├────────────────
│     │ var.env is "prod"
│     │ var.replicas is 1
│
│ Το prod θέλει τουλάχιστον 2 replicas, ώστε να αντέχει την πτώση ενός.
│
│ This was checked by the validation rule at variables.tf:36,3-13.
╵
```

Το σφάλμα δείχνει τώρα δύο τιμές, το `var.env` και το `var.replicas`, γιατί το condition διαβάζει και τις δύο. Δώσε τρία αντίγραφα και ο κανόνας περνά.

TERMINAL

```
terraform apply -auto-approve -var env=prod -var replicas=3
```

TERMINAL

```
cat web-prod.conf
```

ΕΞΟΔΟΣ

```
service = web
env = prod
replicas = 3
```

Η αναφορά σε άλλη μεταβλητή επιτρέπεται από το Terraform 1.9. Παλαιότερες εκδόσεις δέχονταν μόνο τη μεταβλητή που ελέγχεται και απέρριπταν τον κανόνα με `Invalid reference in variable validation`. Η εξέταση βασίζεται στην 1.12, οπότε η δυνατότητα είναι μέσα στην ύλη. Το condition μπορεί να διαβάζει και locals ή data sources του ίδιου module. Πρέπει όμως να αναφέρει πάντα και τη δική του μεταβλητή. Ένας κανόνας για το `replicas` που δεν περιέχει το `var.replicas` απορρίπτεται με το μήνυμα ότι πρέπει να αναφέρει το `var.replicas`.

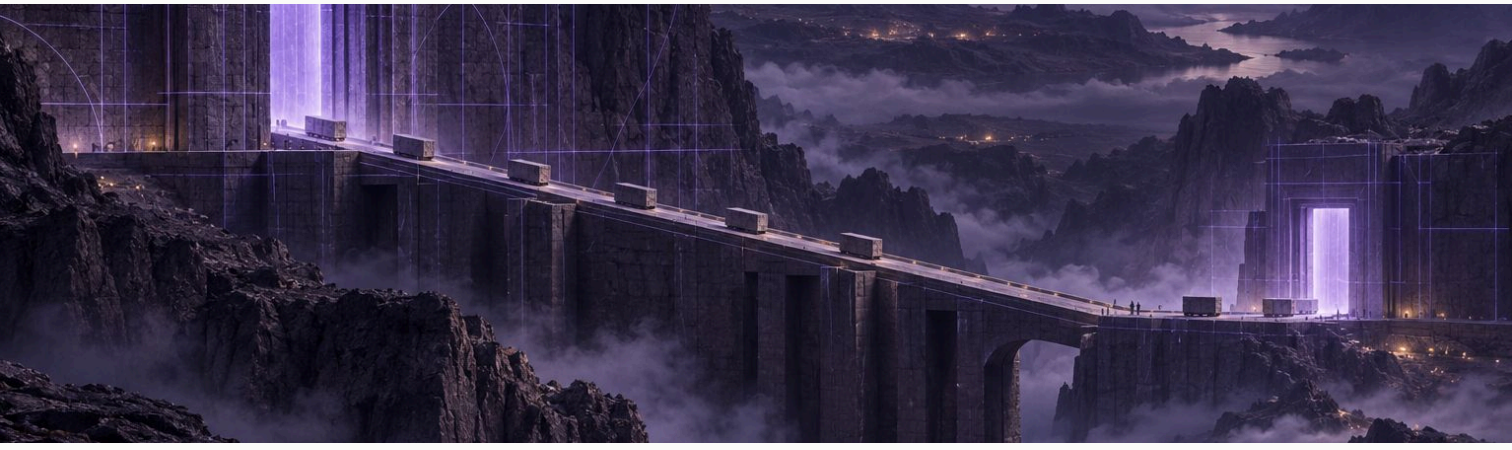
Καλά μηνύματα λάθους

Το `error_message` το διαβάζει κάποιος που δεν βλέπει τον κώδικά σου. Ίσως ένας συνάδελφος σε log ενός pipeline, ίσως εσύ μετά από έξι μήνες. Ένα καλό μήνυμα κάνει τρία πράγματα.

- Λέει ποιο input έχει πρόβλημα, με το όνομά του.
- Λέει τι περιμένει ο κανόνας, όχι μόνο ότι κάτι πήγε στραβά.
- Είναι πλήρης πρόταση, με κεφαλαίο στην αρχή και τελεία στο τέλος.

Κακό μήνυμα	Καλό μήνυμα
`Invalid value.`	`Το env πρέπει να είναι dev, staging ή prod.`
`Λάθος μήκος`	`Το service έχει έως 12 χαρακτήρες. Το ${var.service} έχει ${length(var.service)}.`
`replicas`	`Το prod θέλει τουλάχιστον 2 replicas, ώστε να αντέχει την πτώση ενός.`

Το Terraform δεν ελέγχει το ύφος. Δέχεται ακόμη και ένα `error_message` με πεζά χωρίς τελεία. Η πλήρης πρόταση είναι σύμβαση της τεκμηρίωσης της HashiCorp και των δημόσιων modules.



Preconditions και postconditions

Η validation βλέπει μόνο τις τιμές εισόδου. Πολλές υποθέσεις όμως αφορούν πράγματα που δεν τα δίνει ο χρήστης. Ένα αρχείο που διαβάζεις με data source, μια τιμή που υπολογίζει ένας provider, ένα attribute ενός άλλου resource. Για αυτά υπάρχουν τα preconditions και τα postconditions.

Το block lifecycle

Το κεφάλαιο 11 ανέφερε το `lifecycle` ανάμεσα στα meta-arguments. Είναι ένα nested block που δέχεται κάθε resource και ρυθμίζει πώς το χειρίζεται το ίδιο το Terraform. Τα περισσότερα arguments του, όπως το `create_before_destroy` και το `prevent_destroy`, είναι το θέμα του κεφαλαίου 14. Εδώ χρησιμοποιείς δύο nested blocks του, το `precondition` και το `postcondition`. Μπαίνουν και στο `lifecycle` ενός data source. Ένα resource μπορεί να έχει όσα θέλει από το καθένα.

precondition

Μια άλλη ομάδα κρατά τις ρυθμίσεις της εφαρμογής στο `settings.json`. Η configuration σου το διαβάζει και γράφει από αυτό το `app.conf`. Η εφαρμογή τρέχει χωρίς δικαιώματα root, οπότε δεν μπορεί να ανοίξει θύρα κάτω από το 1024. Αυτή είναι υπόθεση για ένα αρχείο που αλλάζει κάποιος άλλος, όχι για μια μεταβλητή.

TERMINAL

```
mkdir ../settings
cd ../settings
cp ../app/versions.tf .
```

settings.json

```
{
  "port": 80,
  "replicas": 2
}
```

main.tf

```
data "local_file" "settings" {
  filename = "settings.json"
}

locals {
  settings = jsondecode(data.local_file.settings.content)
}

resource "local_file" "conf" {
  filename = "app.conf"
  content  = "port = ${local.settings.port}\nreplicas = ${local.settings.replicas}\n"

  lifecycle {
    precondition {
      condition     = local.settings.port >= 1024
      error_message = "Η θύρα ${local.settings.port} του settings.json θέλει root. Βάλε από 1024 και πάνω."
    }
  }
}
```

Το data source `local_file` του κεφαλαίου 12 διαβάζει το αρχείο και η `jsondecode` το κάνει object. Το `precondition` μπαίνει μέσα στο `lifecycle` του resource που χρειάζεται την υπόθεση. Έτσι όποιος διαβάζει το `local_file.conf` βλέπει αμέσως τι προϋποθέτει.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
data.local_file.settings: Reading...
data.local_file.settings: Read complete after 0s [id=7ba78cf9f5e5e98200cdfdc23b5a3593d08846e1]

Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Resource precondition failed
│
│   on main.tf line 15, in resource "local_file" "conf":
│   15:       condition     = local.settings.port >= 1024
│      ├────────────────
│      │ local.settings.port is 80
│
│ Η θύρα 80 του settings.json θέλει root. Βάλε από 1024 και πάνω.
╵
```

Πρώτα διαβάστηκε το data source. Η τιμή της θύρας έγινε γνωστή και το Terraform έλεγξε το precondition πριν σχεδιάσει την αλλαγή του `local_file.conf`. Το σφάλμα είναι τώρα `Resource precondition failed` και δείχνει την τιμή `local.settings.port is 80`. Το plan σταμάτησε, οπότε κανένα resource δεν άλλαξε. Διόρθωσε το αρχείο και ξανατρέξε.

settings.json

```
{
  "port": 8020,
  "replicas": 2
}
```

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
cat app.conf
```

ΕΞΟΔΟΣ

```
port = 8020
replicas = 2
```

Ένα precondition ελέγχεται πάντα πριν από την αλλαγή του αντικειμένου του. Όταν οι τιμές που διαβάζει είναι γνωστές στο plan, όπως εδώ, ελέγχεται στο plan. Όταν εξαρτώνται από κάτι που θα

γίνει γνωστό μόνο στο apply, το Terraform περιμένει. Το ελέγχει στο apply, ακριβώς πριν φτιάξει ή αλλάξει το resource. Και στις δύο περιπτώσεις ένα precondition που αποτυγχάνει αφήνει το resource ανέγγιχτο. Η άσκηση 13.5 σου δείχνει τη δεύτερη περίπτωση.

Μέσα σε precondition δεν υπάρχει `self`. Το αντικείμενο δεν έχει φτιαχτεί ακόμη, οπότε δεν έχεις τι να διαβάσεις από αυτό. Αν το γράψεις, το Terraform απαντά `Invalid "self" reference`.

postcondition και self

Το postcondition ελέγχει το αποτέλεσμα. Ταιριάζει σε τιμές που δεν τις ξέρεις πριν φτιαχτεί το αντικείμενο, γιατί τις υπολογίζει ο provider. Φτιάξε νέο φάκελο, που χρειάζεται και τον `random`.

TERMINAL

```
mkdir ../ports
cd ../ports
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

main.tf

```
variable "service" {
  type        = string
  description = "Το όνομα της υπηρεσίας"
  default     = "web"
}

resource "random_integer" "port" {
  min  = 8000
  max  = 8099
  seed = var.service

  lifecycle {
    postcondition {
      condition     = !contains([8080, 8088], self.result)
      error_message = "Η θύρα ${self.result} ανήκει στον proxy. Δώσε άλλο όνομα στο service."
    }
  }
}

resource "local_file" "conf" {
  filename = "${var.service}.conf"
  content  = "port = ${random_integer.port.result}\n"
}
```

Κάθε υπηρεσία παίρνει μια θύρα από το 8000 έως το 8099. Το `seed` του `random_integer` κάνει τον αριθμό σταθερό. Το ίδιο seed δίνει πάντα τον ίδιο αριθμό, οπότε κάθε όνομα υπηρεσίας έχει πάντα τη δική του

θύρα. Δύο θύρες όμως, η 8080 και η 8088, ανήκουν στον proxy. Ποια θύρα θα δώσει ένα όνομα δεν το ξέρεις πριν φτιαχτεί το resource. Ο έλεγχος είναι εγγύηση για το αποτέλεσμα, άρα postcondition.

Το `self` μέσα στο postcondition είναι το ίδιο το αντικείμενο μετά την αλλαγή. Το `self.result` είναι ο αριθμός που βγήκε. Δεν μπορείς να γράψεις `random_integer.port.result` μέσα στο δικό του block. Το Terraform το απορρίπτει με `may not refer to itself`, γι' αυτό υπάρχει το `self`. Κάνε apply για την υπηρεσία `api`.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve -var service=api
```

ΕΞΟΔΟΣ

```
…
random_integer.port: Creating...
random_integer.port: Creation complete after 0s [id=8080]
╷
│ Error: Resource postcondition failed
│
│   on main.tf line 14, in resource "random_integer" "port":
│   14:       condition     = !contains([8080, 8088], self.result)
│     ├────────────────
│     │ self.result is 8080
│
│ Η θύρα 8080 ανήκει στον proxy. Δώσε άλλο όνομα στο service.
╵
```

Η έξοδος κρατά μόνο τις γραμμές του apply και το σφάλμα. Στο plan το `result` ήταν `(known after apply)`, οπότε το postcondition δεν μπορούσε να ελεγχθεί νωρίτερα. Το `random_integer.port` φτιάχτηκε με `id=8080` και αμέσως μετά το postcondition απέτυχε με `Resource postcondition failed`. Τι έμεινε πίσω;

TERMINAL

```
terraform state list
ls
```

ΕΞΟΔΟΣ

```
random_integer.port
main.tf
terraform.tfstate
versions.tf
```

Το `random_integer.port` είναι στο state. Το αντικείμενο φτιάχτηκε στ' αλήθεια, οπότε το Terraform το καταγράφει. Αν το ξεχνούσε, θα έμενε ένα αντικείμενο που δεν διαχειρίζεται κανείς. Το `local_file.conf` όμως δεν φτιάχτηκε και το `api.conf` λείπει. Χρειαζόταν τη θύρα και το Terraform δεν συνεχίζει με ό,τι εξαρτάται από ένα αντικείμενο που απέτυχε τον έλεγχο. Το state κρατά και το αποτέλεσμα του ελέγχου.

TERMINAL

```
jq '.check_results' terraform.tfstate
```

ΕΞΟΔΟΣ

```
[
  {
    "object_kind": "resource",
    "config_addr": "random_integer.port",
    "status": "fail",
    "objects": [
      {
        "object_addr": "random_integer.port",
        "status": "fail",
        "failure_messages": [
          "Η θύρα 8080 ανήκει στον proxy. Δώσε άλλο όνομα στο service."
        ]
      }
    ]
  }
]
```

Αυτό είναι το `check_results` που είδες στο κεφάλαιο 4. Δεν κρατά μόνο τα check blocks. Κρατά το αποτέλεσμα κάθε ελέγχου αυτού του κεφαλαίου, από τις validations έως τα postconditions. Για το `random_integer.port` γράφει `"status": "fail"` και το μήνυμα. Το resource δεν σημειώθηκε ως tainted. Το επόμενο plan δεν θα το αντικαταστήσει μόνο του.

TERMINAL

```
terraform plan -var service=api
```

ΕΞΟΔΟΣ

```
random_integer.port: Refreshing state... [id=8080]

Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Resource postcondition failed
│
│   on main.tf line 14, in resource "random_integer" "port":
│   14:       condition     = !contains([8080, 8088], self.result)
│     ├────────────────
│     │ self.result is 8080
│
│ Η θύρα 8080 ανήκει στον proxy. Δώσε άλλο όνομα στο service.
╵
```

Τώρα το postcondition απέτυχε στο plan, μετά το refresh. Ο αριθμός είναι πλέον γνωστός από το state, οπότε το Terraform τον ελέγχει αμέσως. Δεν μπορείς να προχωρήσεις μέχρι να διορθώσεις την αιτία. Εδώ η αιτία είναι το όνομα. Τρέξε apply με το `default`, το `web`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
      ~ seed   = "api" -> "web" # forces replacement
…
Plan: 2 to add, 0 to change, 1 to destroy.
…
Apply complete! Resources: 2 added, 0 changed, 1 destroyed.
```

TERMINAL

```
cat web.conf
```

ΕΞΟΔΟΣ

```
port = 8020
```

Η αλλαγή του `seed` προκάλεσε αντικατάσταση. Το νέο αντικείμενο πήρε τη θύρα 8020, το postcondition πέρασε και το `local_file.conf` φτιάχτηκε.

ΚΡΑΤΑ ΑΥΤΟ

Ένα postcondition που αποτυγχάνει μετά τη δημιουργία βγάζει error. Το αντικείμενο μένει στο state χωρίς να γίνει tainted. Ό,τι εξαρτάται από αυτό δεν φτιάχνεται και κάθε επόμενο plan αποτυγχάνει μέχρι να διορθώσεις την αιτία.

Ένα postcondition δεν περιμένει πάντα το apply. Αν όλες οι τιμές που διαβάζει είναι γνωστές στο plan, για παράδειγμα ένα attribute που όρισες εσύ, ελέγχεται στο plan με τις τιμές που σχεδιάζονται. Το ίδιο ισχύει για ένα postcondition σε data source, που διαβάζεται στο plan.

precondition σε output

Ένα block `output` δέχεται `precondition` απευθείας, χωρίς `lifecycle`. Δεν δέχεται `postcondition`. Γύρνα στον φάκελο `settings` και πρόσθεσε ένα output με τη διεύθυνση της εφαρμογής.

TERMINAL

```
cd ../settings
```

outputs.tf

```
output "url" {
  value = "http://${local.settings.host}:${local.settings.port}"

  precondition {
    condition     = can(local.settings.host)
    error_message = "Το settings.json χρειάζεται πεδίο host με το όνομα του μηχανήματος."
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
data.local_file.settings: Reading...
data.local_file.settings: Read complete after 0s [id=f53988a7361a81aa0ac029297363023c12b204a1]
local_file.conf: Refreshing state... [id=b79fdafb2f00e88ee36031b9dc35d72b243e6764]

Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Module output value precondition failed
│
│   on outputs.tf line 5, in output "url":
│    5:     condition     = can(local.settings.host)
│     ├────────────────
│     │ local.settings is object with 2 attributes
│
│ Το settings.json χρειάζεται πεδίο host με το όνομα του μηχανήματος.
╵
```

Το `settings.json` δεν έχει `host`. Το Terraform ελέγχει το precondition πριν υπολογίσει το `value`, οπότε βλέπεις μόνο το δικό σου μήνυμα, κάτω από το `Module output value precondition failed`. Χωρίς το precondition θα έβλεπες `Unsupported attribute` με την αγγλική εξήγηση ότι το object δεν έχει attribute `host`. Σωστό, αλλά δεν λέει τι να κάνεις.

settings.json

```
{
  "host": "localhost",
  "port": 8020,
  "replicas": 2
}
```

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
terraform output url
```

ΕΞΟΔΟΣ

```
"http://localhost:8020"
```

Το precondition σε output έχει τη μεγαλύτερη αξία στα modules του κεφαλαίου 15. Εκεί ένα module υπόσχεται κάτι σε όποιον το καλεί και το precondition εγγυάται ότι η υπόσχεση τηρείται.

check blocks

Τα τρία προηγούμενα σταματούν τη δουλειά. Μερικές φορές αυτό είναι υπερβολικό. Θέλεις να μάθεις ότι κάτι δεν πάει καλά, χωρίς να μπλοκάρεις ένα apply που μπορεί να διορθώνει κάτι άλλο. Γι' αυτό υπάρχει το block `check`.



assert που δεν σταματά

Γύρνα στον φάκελο `ports`. Βγάλε το postcondition και γράψε τον ίδιο έλεγχο ως check block.

TERMINAL

```
cd ../ports
```

main.tf

```
variable "service" {
  type        = string
  description = "Το όνομα της υπηρεσίας"
  default     = "web"
}

resource "random_integer" "port" {
  min  = 8000
  max  = 8099
  seed = var.service
}

resource "local_file" "conf" {
  filename = "${var.service}.conf"
  content  = "port = ${random_integer.port.result}\n"
}

check "port" {
  assert {
    condition     = !contains([8080, 8088], random_integer.port.result)
    error_message = "Η θύρα ${random_integer.port.result} ανήκει στον proxy. Δώσε άλλο όνομα στο service."
  }
}
```

Το `check` είναι block πρώτου επιπέδου, δίπλα στα resources, με ένα label για όνομα. Μέσα έχει ένα ή περισσότερα blocks `assert`, το καθένα με `condition` και `error_message`. Δεν ανήκει σε κανένα resource, οπότε δεν έχει `self`. Διαβάζει ό,τι θέλει από το module με κανονικές αναφορές. Δες πρώτα το plan για την υπηρεσία `api`.

TERMINAL

```
terraform plan -var service=api
```

ΕΞΟΔΟΣ

```
…
Plan: 2 to add, 0 to change, 2 to destroy.

╷
│ Warning: Check block assertion known after apply
│
│   on main.tf line 20, in check "port":
│   20:     condition     = !contains([8080, 8088], random_integer.port.result)
│      ├────────────────
│      │ random_integer.port.result is a number
│
│ The condition could not be evaluated at this time, a result will be known
│ when this plan is applied.
╵

…
```

Η αλλαγή του `seed` αντικαθιστά το `random_integer.port`, οπότε ο νέος αριθμός είναι άγνωστος στο plan. Ο έλεγχος δεν μπορεί να γίνει ακόμη και το Terraform το λέει με την προειδοποίηση `Check block assertion known after apply`. Κάνε apply.

TERMINAL

```
terraform apply -auto-approve -var service=api
```

ΕΞΟΔΟΣ

```
random_integer.port: Refreshing state... [id=8020]
local_file.conf: Refreshing state... [id=4743f14a3b79ea2f74038b9b2e86ed8c17bace2a]
…
local_file.conf: Destroying... [id=4743f14a3b79ea2f74038b9b2e86ed8c17bace2a]
local_file.conf: Destruction complete after 0s
random_integer.port: Destroying... [id=8020]
random_integer.port: Destruction complete after 0s
random_integer.port: Creating...
random_integer.port: Creation complete after 0s [id=8080]
local_file.conf: Creating...
local_file.conf: Creation complete after 0s [id=c4a8ceac89382686b7220243d97f08299b545f08]

╷
│ Warning: Check block assertion failed
│
│   on main.tf line 20, in check "port":
│   20:     condition     = !contains([8080, 8088], random_integer.port.result)
│      ├────────────────
│      │ random_integer.port.result is 8080
│
│ Η θύρα 8080 ανήκει στον proxy. Δώσε άλλο όνομα στο service.
╵

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

Το apply αντικατέστησε το `random_integer.port`, πήρε τη θύρα 8080 και έφτιαξε το `api.conf`. Μετά ελέγχθηκε το check. Απέτυχε ως `Warning` και όχι ως `Error`. Το apply ολοκληρώθηκε κανονικά.

TERMINAL

```
cat api.conf
```

ΕΞΟΔΟΣ

```
port = 8080
```

Το αρχείο υπάρχει με τη λάθος θύρα. Αυτό είναι το αντίτιμο του check block. Σε ενημερώνει, αλλά δεν σε προστατεύει. Κάθε επόμενο plan και apply θα ξαναβγάζει την ίδια προειδοποίηση, μέχρι να διορθωθεί η αιτία.

Scoped data source

Ένα check block μπορεί να έχει μέσα του ένα block `data`. Λέγεται **scoped data source**, γιατί υπάρχει μόνο μέσα στο check. Το διαβάζουν τα `assert` του ίδιου block και τίποτε άλλο. Αν ένα output το αναφέρει, το `validate` απαντά `Reference to scoped resource`. Ένα check block δέχεται το πολύ ένα τέτοιο data source.

Στο AWS ένα scoped data source κάνει συνήθως ένα αίτημα HTTP στη σελίδα που μόλις στήθηκε. Εδώ η εφαρμογή γράφει την κατάστασή της στο `status.txt` και το check το διαβάζει.

TERMINAL

```
mkdir ../health
cd ../health
cp ../app/versions.tf .
```

main.tf

```
resource "local_file" "conf" {
  filename = "app.conf"
  content  = "port = 8020\n"
}

check "health" {
  data "local_file" "status" {
    filename = "status.txt"
  }

  assert {
    condition     = trimspace(data.local_file.status.content) == "ok"
    error_message = "Η εφαρμογή αναφέρει ${trimspace(data.local_file.status.content)} αντί για ok."
  }
}
```

Το `status.txt` δεν υπάρχει ακόμη. Ένα κανονικό data source που δεν βρίσκει το αρχείο του σταματά το plan με error.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.local_file.status: Reading...
…
local_file.conf: Creating...
local_file.conf: Creation complete after 0s [id=4743f14a3b79ea2f74038b9b2e86ed8c17bace2a]
data.local_file.status: Reading...
…
│ Warning: Read local file data source error
…
│ +Original Error: open status.txt: no such file or directory
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το data source διαβάστηκε δύο φορές. Μία στο plan και μία στο τέλος του apply, αφού φτιάχτηκε το `local_file.conf`. Μέσα σε check block το σφάλμα του έγινε προειδοποίηση, `Read local file data source error`. Το apply τελείωσε κανονικά. Γράψε τώρα μια κακή κατάσταση και τρέξε plan.

TERMINAL

```
echo degraded > status.txt
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # data.local_file.status will be read during apply
  # (config will be reloaded to verify a check block)
…
Plan: 0 to add, 0 to change, 0 to destroy.
╷
│ Warning: Check block assertion failed
│
│   on main.tf line 12, in check "health":
│   12:     condition     = trimspace(data.local_file.status.content) == "ok"
│     ├────────────────
│     │ data.local_file.status.content is "degraded\n"
│
│ Η εφαρμογή αναφέρει degraded αντί για ok.
╵
…
```

Τώρα το αρχείο διαβάστηκε και το `assert` απέτυχε με `Check block assertion failed`. Το μήνυμα δείχνει την τιμή `degraded`. Το plan δείχνει επίσης ότι το `data.local_file.status` θα διαβαστεί ξανά στο apply. Το Terraform ξαναδιαβάζει ένα scoped data source μετά τις αλλαγές, ώστε ο έλεγχος να βλέπει την υποδομή όπως έμεινε. Η περίληψη λέει `0 to add, 0 to change, 0 to destroy`. Κανένα resource δεν αλλάζει.

TERMINAL

```
echo ok > status.txt
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

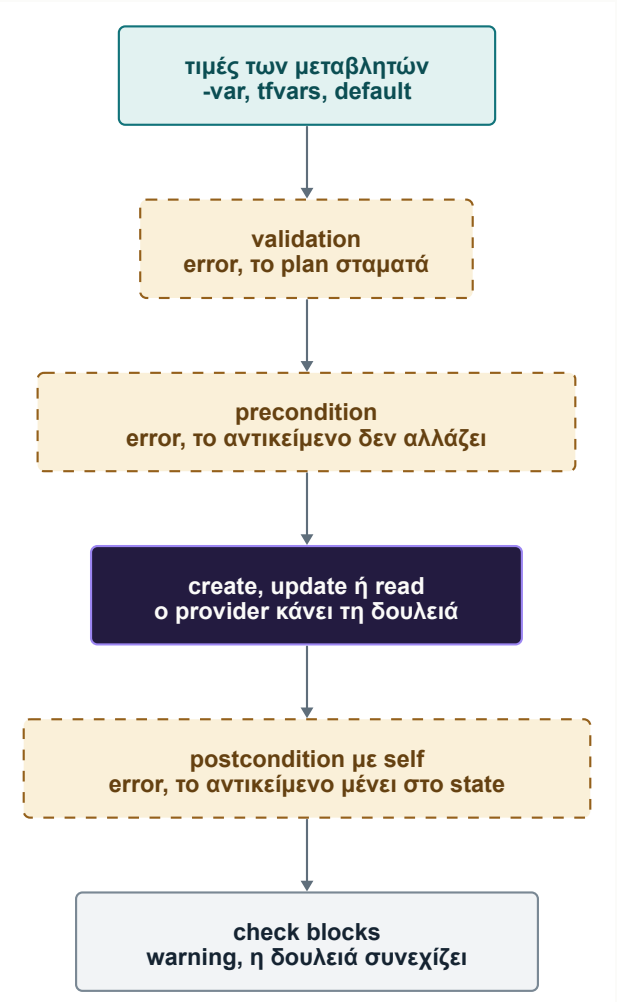
Με `ok` το check περνά και η έξοδος δεν έχει καμία προειδοποίηση.

ΣΗΜΕΙΩΣΗ

Επειδή το scoped data source διαβάζεται πάντα ξανά στο apply, ένα `terraform plan -detailed-exitcode` του κεφαλαίου 3 επιστρέφει `2` ακόμη κι όταν κανένα resource δεν αλλάζει. Αν ένα pipeline χρησιμοποιεί τον κωδικό για να βρει drift, πρέπει να το ξέρει.

Τα check blocks ελέγχονται σε κάθε plan και σε κάθε apply, στο τέλος, αφού σχεδιαστούν ή γίνουν οι αλλαγές. Γι' αυτό λειτουργούν ως συνεχής παρακολούθηση. Το HCP Terraform του κεφαλαίου 26 τα τρέχει και περιοδικά, χωρίς apply, στα health assessments.

Ποιον μηχανισμό διαλέγεις



Σχήμα 13.1 · Η σειρά των ελέγχων σε ένα apply. Οι τρεις πρώτοι βγάζουν error και σταματούν τη δουλειά. Το check block έρχεται τελευταίο και βγάζει μόνο warning.

Μηχανισμός	Πού γράφεται	Πότε ελέγχεται	Όταν αποτύχει
`validation`	Μέσα σε `variable`	Στο plan, μόλις η μεταβλητή πάρει τιμή	Error, το plan σταματά
`precondition`	Στο `lifecycle` ενός resource ή data source, μέσα σε `output`	Πριν από την αλλαγή, στο plan ή στο apply	Error, το αντικείμενο δεν αλλάζει
`postcondition`	Στο `lifecycle` ενός resource ή data source	Μετά την αλλαγή ή την ανάγνωση	Error, το αντικείμενο μένει στο state
`check`	Block πρώτου επιπέδου με `assert`	Στο τέλος κάθε plan και apply	Warning, η δουλειά συνεχίζει

Η επιλογή ξεκινά από την ερώτηση «τι ελέγχω».

- Μια τιμή που δίνει ο χρήστης, όπως όνομα, μέγεθος ή περιβάλλον. Γράψε validation στη μεταβλητή. Πιάνει το λάθος όσο πιο νωρίς γίνεται.
- Μια υπόθεση για κάτι που υπάρχει ήδη ή που φτιάχνει άλλο resource, πριν φτιάξεις αυτό που τη χρειάζεται. Γράψε precondition στο resource που θα χαλάσει αν η υπόθεση δεν ισχύει.
- Μια εγγύηση για το αποτέλεσμα, που τη μαθαίνεις μόνο αφού φτιαχτεί το αντικείμενο. Γράψε postcondition με `self`.
- Την υγεία της υποδομής στο σύνολό της, χωρίς να μπλοκάρεις κανένα apply. Γράψε check block, με scoped data source αν πρέπει να διαβάσεις κάτι ζωντανά.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση δίνει συχνά ένα σενάριο και ρωτά ποιος μηχανισμός ταιριάζει. Θυμήσου πού ανήκει ο καθένας. Η validation ανήκει στο `variable`, ενώ το precondition και το postcondition στο `lifecycle`. Το `self` υπάρχει μόνο στο postcondition. Το check block είναι ο μόνος μηχανισμός που βγάζει warning αντί για error, οπότε δεν σταματά ποτέ ένα plan ή ένα apply.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch13`. Γράψε τα αρχεία που ζητά η εκφώνηση, τρέξε `terraform init` και μετά τις εντολές της άσκησης. Χρησιμοποίησε τις εκδόσεις providers του κεφαλαίου.

ΑΣΚΗΣΗ 13.1 Επιτρεπτές περιοχές

Γράψε configuration με τον provider `local` και μια μεταβλητή `region` τύπου `string` με `default` το `eu-central-1`. Η μεταβλητή δέχεται μόνο τις τιμές `eu-central-1` και `eu-west-1`. Ένα `local_file` γράφει την περιοχή στο `region.txt`. Τρέξε `terraform plan -var region=us-east-1`. Το αναμενόμενο αποτέλεσμα είναι το σφάλμα. Το δικό σου μήνυμα θα διαφέρει στη διατύπωση.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
│ Error: Invalid value for variable
│
│   on main.tf line 10:
│   10: variable "region" {
│     ├────────────────
│     │ var.region is "us-east-1"
│
│ Το region πρέπει να είναι eu-central-1 ή eu-west-1.
│
│ This was checked by the validation rule at main.tf:15,3-13.
```

ΑΣΚΗΣΗ 13.2 Θύρα σε εύρος

Γράψε configuration με τον provider `local` και μια μεταβλητή `port` τύπου `number` με `default` το `8080`. Πρόσθεσε δύο validation blocks. Το πρώτο απαιτεί ακέραιο αριθμό και το δεύτερο τιμή από 1024 έως 65535. Ένα `local_file` γράφει τη θύρα στο `port.txt`. Τρέξε plan με `-var port=80.5`. Το αναμενόμενο αποτέλεσμα δείχνει τις γραμμές των σφαλμάτων και τα μηνύματα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Invalid value for variable
╷
│ Η θύρα πρέπει να είναι ακέραιος αριθμός.
╷
│ Error: Invalid value for variable
╷
│ Η θύρα πρέπει να είναι από 1024 έως 65535.
╵
```

ΑΣΚΗΣΗ 13.3 Το validate δεν βλέπει τιμές

Γράψε σε νέο φάκελο το `main.tf` που ακολουθεί. Πριν τρέξεις οτιδήποτε, πρόβλεψε τι θα πουν το `terraform validate` και το `terraform plan`. Μετά τρέξε `terraform init` και τις δύο εντολές με αυτή τη σειρά.

main.tf

```
variable "size" {
  type    = number
  default = 0

  validation {
    condition     = var.size > 0
    error_message = "Το size πρέπει να είναι θετικός αριθμός."
  }
}

output "size" {
  value = var.size
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Success! The configuration is valid.
╷
│ Error: Invalid value for variable
╷
│ Το size πρέπει να είναι θετικός αριθμός.
╵
```

ΑΣΚΗΣΗ 13.4 Το μήνυμα που δεν εμφανίζεται

Γράψε σε νέο φάκελο το `main.tf` που ακολουθεί και τρέξε `terraform plan -var bucket=My_Bucket`. Το σφάλμα δεν δείχνει το `error_message`. Διόρθωσε το condition ώστε ο χρήστης να βλέπει το δικό σου μήνυμα και ξανατρέξε την ίδια εντολή.

main.tf

```
variable "bucket" {
  type = string

  validation {
    condition     = regex("^[a-z0-9.-]+$", var.bucket) == var.bucket
    error_message = "Το bucket έχει μόνο πεζά γράμματα, ψηφία, τελείες και παύλες."
  }
}

output "bucket" {
  value = var.bucket
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Invalid value for variable
╵
│    1: variable "bucket" {
╷
│     ├─ var.bucket is "My_Bucket"
╵
│ Το bucket έχει μόνο πεζά γράμματα, ψηφία, τελείες και παύλες.
╵
```

ΑΣΚΗΣΗ 13.5 Precondition στο apply

Γράψε configuration με τους providers `local` και `random`. Ένα `random_integer` με όνομα `port` δίνει αριθμό από 8000 έως 8099 με `seed = "api"`. Ένα `local_file` με όνομα `conf` γράφει τη θύρα στο `app.conf` και έχει precondition ότι η θύρα δεν είναι η 8080. Πρόβλεψε αν ο έλεγχος θα αποτύχει στο plan ή στο apply. Τρέξε `terraform apply -auto-approve` και μετά `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
random_integer.port
```

ΑΣΚΗΣΗ 13.6 Postcondition στο plan

Γράψε configuration με τον provider `local` και ένα `local_file` με όνομα `secret` που γράφει το `secret.txt`. Πρόσθεσε postcondition ότι το `file_permission` του αρχείου είναι `0600`, χωρίς να ορίσεις το `file_permission`. Τρέξε `terraform plan`. Αποτυγχάνει στο plan ή θα περίμενε το apply; Μετά διόρθωσε το resource ώστε να περνά και κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Resource postcondition failed
╵
│     | self.file_permission is "0777"
╷
│ Το secret.txt έχει δικαιώματα 0777. Όρισε file_permission = "0600".
╵
```

ΑΣΚΗΣΗ 13.7 Check μετά το apply

Γράψε configuration με τον provider `local`. Ένα `local_file` με όνομα `motd` γράφει στο `motd.txt` το κείμενο `Καλώς ήρθες στον server` με αλλαγή γραμμής στο τέλος. Ένα check block με όνομα `motd` διαβάζει το αρχείο με scoped data source και ελέγχει ότι έχει έως 20 χαρακτήρες. Πάρε το `filename` του data source από το `local_file.motd`, ώστε να διαβαστεί μετά τη δημιουργία. Κάνε apply και δες αν ολοκληρώθηκε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Warning: Check block assertion failed
╵
│ Το motd.txt έχει 24 χαρακτήρες. Το όριο είναι 20.
╵
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 13.1

Η μεταβλητή `instance_type` πρέπει να είναι `t3.micro` ή `t3.small`. Θέλεις μια λάθος τιμή να απορρίπτεται με error πριν τη χρησιμοποιήσει οποιοδήποτε resource. Τι χρησιμοποιείς;

a. Ένα check block με assert για το `var.instance_type`.
b. Ένα block `validation` μέσα στο block `variable`.
c. Ένα postcondition στο resource που χρησιμοποιεί τη μεταβλητή.
d. Ένα precondition σε output με την τιμή της μεταβλητής.

ΕΡΩΤΗΣΗ 13.2

Ένα assert σε check block αποτυγχάνει κατά το `terraform apply`. Τι κάνει το Terraform;

a. Σταματά το apply και αναιρεί όσες αλλαγές έγιναν.
b. Σημειώνει ως tainted τα resources που αναφέρει το assert.
c. Ολοκληρώνει το apply και εμφανίζει warning με το μήνυμα του assert.
d. Σταματά πριν από κάθε αλλαγή και βγάζει error.

ΕΡΩΤΗΣΗ 13.3

Ένα resource φτιάχνεται κατά το apply και αμέσως μετά το postcondition του αποτυγχάνει. Τι ισχύει;

a. Το apply αναφέρει error, το αντικείμενο μένει στο state και όσα εξαρτώνται από αυτό δεν φτιάχνονται.
b. Το Terraform σβήνει αυτόματα το αντικείμενο, ώστε το state να μείνει καθαρό.
c. Το αντικείμενο σημειώνεται ως tainted και αντικαθίσταται στο επόμενο apply.
d. Το apply βγάζει warning και συνεχίζει με τα υπόλοιπα resources.

ΕΡΩΤΗΣΗ 13.4

Πού μπορείς να γράψεις ένα block `precondition`; Επίλεξε δύο.

a. Μέσα σε block `variable`
b. Μέσα σε block `output`
c. Μέσα στο `lifecycle` ενός resource
d. Μέσα σε block `provider`
e. Μέσα στο block `terraform`

ΕΡΩΤΗΣΗ 13.5

Σωστό ή λάθος; Στο Terraform 1.12 το `condition` ενός `validation` στη μεταβλητή `max_size` μπορεί να συγκρίνει το `var.max_size` με το `var.min_size`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 13.6

Μια μεταβλητή έχει `default = 0` και validation με `condition = var.size > 0`. Τρέχεις `terraform validate` και μετά `terraform plan` χωρίς άλλες τιμές. Τι συμβαίνει;

a. Το `validate` αποτυγχάνει, γιατί το `default` παραβιάζει τον κανόνα.
b. Και οι δύο εντολές πετυχαίνουν, γιατί το `default` δεν ελέγχεται ποτέ.
c. Το `plan` ρωτά για νέα τιμή, γιατί το `default` είναι άκυρο.
d. Το `validate` πετυχαίνει και το `plan` αποτυγχάνει με `Invalid value for variable`.



ΚΕΦΑΛΑΙΟ 14

Lifecycle και αντικατάσταση

Ως τώρα το Terraform αποφάσιζε μόνο του πώς αλλάζει ένα resource. Σε αυτό το κεφάλαιο παίρνεις τον έλεγχο με το block `lifecycle`. Αλλάζεις τη σειρά μιας αντικατάστασης ώστε η υπηρεσία να μη μείνει χωρίς server, προστατεύεις ένα resource από διαγραφή, αγνοείς αλλαγές που κάνει κάποιος άλλος και ζητάς αντικατάσταση όταν αλλάζει ένα άλλο resource. Στο τέλος βλέπεις πότε ένα resource γίνεται tainted και πώς το διορθώνεις.

Το block lifecycle

Στο κεφάλαιο 13 έβαλες `precondition` και `postcondition` σε ένα block `lifecycle` μέσα σε resource. Το ίδιο block δέχεται και τέσσερις ρυθμίσεις που αλλάζουν τον τρόπο με τον οποίο το Terraform φτιάχνει, αλλάζει και σβήνει ένα resource.

HCL

```
resource "terraform_data" "example" {
  input = "a"

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [input]
    replace_triggered_by  = [terraform_data.other]
  }
}
```

Το `lifecycle` είναι meta-argument, όπως το `depends_on` του κεφαλαίου 6 και τα `count` και `for_each` του κεφαλαίου 11. Το διαβάζει το ίδιο το Terraform και όχι ο provider, οπότε δουλεύει σε κάθε τύπο resource. Το Terraform διαβάζει τις τέσσερις ρυθμίσεις ήδη όταν φορτώνει τα αρχεία, πριν υπολογίσει μεταβλητές ή άλλες εκφράσεις. Γι' αυτό τα `create_before_destroy` και `prevent_destroy` δέχονται μόνο σκέτο `true` ή `false`. Τα `ignore_changes` και `replace_triggered_by` δέχονται λίστες με ονόματα attributes ή αναφορές σε resources, όπως θα δεις παρακάτω.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Στις ρυθμίσεις του `lifecycle` δεν γράφεις μεταβλητές, locals ή συναρτήσεις. Ένα `prevent_destroy = var.protect` σταματά ήδη στο `init` με το σφάλμα `Variables not allowed`. Μια συνάρτηση δίνει `Function calls not allowed`. Δεν μπορείς λοιπόν να προστατεύεις ένα resource μόνο στο production με μια μεταβλητή.

Γιατί πρώτα destroy

Στο κεφάλαιο 3 είδες ότι ένα argument που δεν αλλάζει επί τόπου οδηγεί σε αντικατάσταση με σύμβολο `-/+`. Το Terraform σβήνει πρώτα το παλιό αντικείμενο και μετά φτιάχνει το νέο. Η σειρά αυτή είναι η ασφαλής προεπιλογή. Πολλά πραγματικά αντικείμενα έχουν μοναδικό όνομα, μια σταθερή διεύθυνση IP ή ένα αρχείο σε συγκεκριμένο path. Δύο αντικείμενα με το ίδιο όνομα δεν μπορούν να υπάρξουν μαζί, οπότε το create πριν από το destroy θα αποτύγχανε. Το destroy πρώτα δουλεύει πάντα.

Το τίμημα είναι ένα κενό. Από τη στιγμή που σβήνεται το παλιό μέχρι να είναι έτοιμο το νέο δεν υπάρχει τίποτα. Για έναν server αυτό σημαίνει διακοπή υπηρεσίας για όσο κρατά η δημιουργία, συχνά λεπτά.

Για να δεις το κενό χρειάζεσαι ένα resource που αργεί να φτιαχτεί. Ο provider `hashicorp/time` έχει το `time_sleep`, που περιμένει όσο λέει το `create_duration` πριν δηλώσει ότι δημιουργήθηκε. Δεν φτιάχνει τίποτα άλλο. Εδώ παίζει τον ρόλο ενός server που χρειάζεται τρία δευτερόλεπτα για να ξεκινήσει. Το argument `triggers` είναι ένα map. Κάθε αλλαγή σε αυτό αντικαθιστά το resource, όπως το `triggers_replace` του `terraform_data`.

TERMINAL

```
mkdir swap
cd swap
```

versions.tf

```
terraform {
  required_providers {
    time = {
      source  = "hashicorp/time"
      version = "0.14.2"
    }
  }
}
```

main.tf

```
variable "image" {
  type    = string
  default = "ubuntu-24.04"
}

resource "time_sleep" "server" {
  create_duration = "3s"

  triggers = {
    image = var.image
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Finding hashicorp/time versions matching "0.14.2"...
- Installing hashicorp/time v0.14.2...
- Installed hashicorp/time v0.14.2 (signed by HashiCorp)
…
```

TERMINAL

```
terraform apply -auto-approve
```

Ο server υπάρχει με το image `ubuntu-24.04`. Άλλαξε τώρα το image από τη γραμμή εντολών, όπως στο κεφάλαιο 7.

TERMINAL

```
terraform apply -auto-approve -var image=ubuntu-26.04
```

ΕΞΟΔΟΣ

```
…
-/+ destroy and then create replacement
…
  # time_sleep.server must be replaced
-/+ resource "time_sleep" "server" {
…
time_sleep.server: Destroying... [id=2026-10-08T02:27:57Z]
time_sleep.server: Destruction complete after 0s
time_sleep.server: Creating...
time_sleep.server: Creation complete after 3s [id=2026-10-08T02:28:01Z]
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Η επικεφαλίδα γράφει `-/+ destroy and then create replacement` και οι γραμμές της εκτέλεσης ακολουθούν αυτή τη σειρά. Πρώτα `Destroying` και `Destruction complete`. Μετά `Creating` και, τρία δευτερόλεπτα

αργότερα, `Creation complete after 3s`. Σε εκείνα τα τρία δευτερόλεπτα δεν υπήρχε κανένας server. Το `id` του `time_sleep` είναι η ώρα δημιουργίας του, γι' αυτό το νέο διαφέρει από το παλιό.

create_before_destroy

Με `create_before_destroy = true` αντιστρέφεις τη σειρά. Το Terraform φτιάχνει πρώτα το νέο αντικείμενο και σβήνει το παλιό μόνο όταν το νέο είναι έτοιμο. Πρόσθεσε το `lifecycle` στο `time_sleep.server`.

main.tf

```
variable "image" {
  type    = string
  default = "ubuntu-24.04"
}

resource "time_sleep" "server" {
  create_duration = "3s"

  triggers = {
    image = var.image
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

TERMINAL

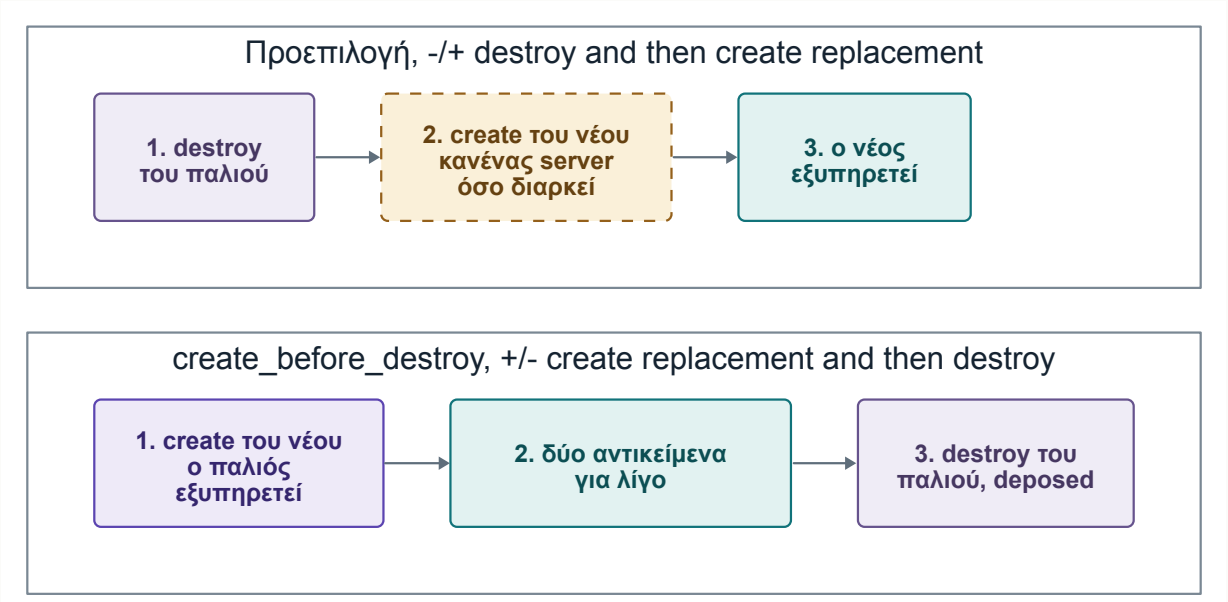
```
terraform apply -auto-approve -var image=debian-13
```

ΕΞΟΔΟΣ

```
…
+/- create replacement and then destroy
…
  # time_sleep.server must be replaced
+/- resource "time_sleep" "server" {
…
Plan: 1 to add, 0 to change, 1 to destroy.
time_sleep.server: Creating...
time_sleep.server: Creation complete after 3s [id=2026-10-08T02:28:04Z]
time_sleep.server (deposed object 0224843d): Destroying... [id=2026-10-08T02:28:01Z]
time_sleep.server: Destruction complete after 0s
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Το σύμβολο είναι τώρα `+/-` και η επικεφαλίδα γράφει `create replacement and then destroy`. Η περίληψη του plan μένει ίδια, ένα `to add` και ένα `to destroy`. Αλλάζει μόνο η σειρά. Το `Creating` έρχεται πρώτο και ο παλιός server ζει όσο ετοιμάζεται ο νέος.

Το παλιό αντικείμενο εμφανίζεται ως `deposed object` με έναν κωδικό. Deposed σημαίνει «εκθρονισμένο». Για λίγο το state κρατά δύο αντικείμενα κάτω από την ίδια διεύθυνση. Το νέο είναι το τρέχον και το παλιό περιμένει να σβηστεί. Αν το destroy του αποτύχει, το deposed αντικείμενο μένει στο state και το επόμενο apply προσπαθεί ξανά να το σβήσει.



Σχήμα 14.1 · Οι δύο σειρές μιας αντικατάστασης. Με την προεπιλογή υπάρχει κενό όσο φτιάχνεται το νέο αντικείμενο. Με create_before_destroy τα δύο αντικείμενα συνυπάρχουν για λίγο.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
>
> Το `-/+` σημαίνει destroy και μετά create, η προεπιλογή. Το `+/-` εμφανίζεται μόνο όταν ισχύει `create_before_destroy` και σημαίνει create και μετά destroy. Το `create_before_destroy` δεν αλλάζει το αν θα γίνει αντικατάσταση. Αλλάζει μόνο τη σειρά.

Δύο αντικείμενα με το ίδιο όνομα

Το `create_before_destroy` προϋποθέτει ότι το παλιό και το νέο αντικείμενο μπορούν να υπάρξουν μαζί. Δες τι γίνεται όταν δεν μπορούν.

TERMINAL

```
mkdir ../names
cd ../names
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

main.tf

```
variable "release" {
  type    = string
  default = "1.0"
}

resource "local_file" "page" {
  filename = "index.html"
  content  = "<h1>Έκδοση ${var.release}</h1>\n"

  lifecycle {
    create_before_destroy = true
  }
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

TERMINAL

```
terraform apply -auto-approve -var release=1.1
```

ΕΞΟΔΟΣ

```
…
local_file.page: Creating...
local_file.page: Creation complete after 0s [id=e02abf2d720e2fd5740b9ac45248c33e13058188]
local_file.page (deposed object 84748b2a): Destroying... [id=b2847dcd58e6738e877ac63748704d9f678b91e0]
local_file.page: Destruction complete after 0s
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
terraform.tfstate.backup
versions.tf
```

Το apply πέτυχε, αλλά το `index.html` λείπει από το `ls`. Το νέο αντικείμενο έγραψε το `index.html` με τη νέα έκδοση. Μετά το destroy του παλιού αντικειμένου έσβησε το αρχείο στο ίδιο path, δηλαδή το καινούργιο. Το επόμενο plan θέλει να φτιάξει ξανά τη σελίδα.

TERMINAL

```
terraform plan -var release=1.1
```

ΕΞΟΔΟΣ

```
…
  # local_file.page will be created
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
```

Στο AWS το ίδιο λάθος φαίνεται αλλιώς. Ένα δεύτερο security group ή bucket με το ίδιο όνομα απορρίπτεται από το API και το apply σταματά με σφάλμα. Η λύση είναι πάντα ένα όνομα που αλλάζει σε κάθε αντικατάσταση, για παράδειγμα με την έκδοση μέσα στο όνομα. Πολλά resources του AWS έχουν γι' αυτό το argument `name_prefix`, που προσθέτει τυχαίο κομμάτι στο όνομα.

ΠΑΓΙΔΑ

Μη βάζεις `create_before_destroy` σε resource με σταθερό μοναδικό όνομα. Η HashiCorp δεν το κάνει προεπιλογή ακριβώς γι' αυτό. Πριν το ενεργοποιήσεις, βεβαιώσου ότι δύο αντικείμενα του τύπου μπορούν να υπάρξουν ταυτόχρονα.

Το create_before_destroy περνά στις εξαρτήσεις

Ο server χρειάζεται ένα image. Γύρνα στο `swap` και πρόσθεσε ένα `terraform_data` που παίζει τον ρόλο του image. Ο server το διαβάζει με αναφορά, οπότε εξαρτάται από αυτό. Το `create_before_destroy` υπάρχει μόνο στον server.

TERMINAL

```
cd ../swap
```

main.tf

```
variable "image" {
  type    = string
  default = "ubuntu-24.04"
}

resource "terraform_data" "image" {
  triggers_replace = var.image
}

resource "time_sleep" "server" {
  create_duration = "3s"

  triggers = {
    image = terraform_data.image.id
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

TERMINAL

```
terraform apply -auto-approve -var image=debian-13
```

Το apply έφτιαξε το image και αντικατέστησε τον server, γιατί το `triggers` του δείχνει πλέον στο `id` του image. Κοίτα τώρα τι έγραψε το Terraform στο state για τα δύο resources. Όπως στο κεφάλαιο 4, το `jq` διαβάζει το αρχείο χωρίς να το αλλάζει.

TERMINAL

```
jq -c '.resources[] | [.type, .name, .instances[0].create_before_destroy]' terraform.tfstate
```

ΕΞΟΔΟΣ

```
["terraform_data","image",true]
["time_sleep","server",true]
```

Και τα δύο έχουν `true`, ενώ στο `terraform_data.image` δεν το έγραψες. Άλλαξε το image και δες τη σειρά.

TERMINAL

```
terraform apply -auto-approve -var image=ubuntu-26.04
```

ΕΞΟΔΟΣ

```
…
+/- create replacement and then destroy
…
+/- resource "terraform_data" "image" {
…
+/- resource "time_sleep" "server" {
…
terraform_data.image: Creating...
…
time_sleep.server: Creating...
…
time_sleep.server (deposed object 33dd6c44): Destroying... [id=2026-10-08T02:28:13Z]
…
terraform_data.image (deposed object 7502fc72): Destroying... [id=d3fe26a8-61cb-8e03-4ca9-26e1fd1c157d]
…
Apply complete! Resources: 2 added, 0 changed, 2 destroyed.
```

Και τα δύο resources έχουν `+/-`. Πρώτα φτιάχτηκε το νέο image, μετά ο νέος server. Ακολούθησε το destroy του παλιού server και τελευταίο το destroy του παλιού image.

Ο λόγος είναι το graph του κεφαλαίου 6. Ο παλιός server ζει μέχρι να είναι έτοιμος ο νέος και χρειάζεται το παλιό image όσο ζει. Το παλιό image λοιπόν δεν μπορεί να σβηστεί πριν από τον παλιό server. Αν το image είχε `-/+`, θα έπρεπε να σβηστεί πρώτο. Οι δύο απαιτήσεις μαζί δίνουν κύκλο. Το Terraform τον αποφεύγει ενεργοποιώντας το `create_before_destroy` σε όλα τα resources από τα οποία εξαρτάται ένα resource με `create_before_destroy`. Η ρύθμιση γράφεται στο state και δεν ακυρώνεται ούτε με `create_before_destroy = false` στο image.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το `create_before_destroy` ενός resource περνά αυτόματα σε όλα τα resources από τα οποία εξαρτάται. Στο plan το βλέπεις ως `+/-` σε resources χωρίς δικό τους `lifecycle`.

prevent_destroy

Κάποια resources δεν πρέπει να σβηστούν ποτέ κατά λάθος, όπως μια βάση δεδομένων με πελάτες. Το `prevent_destroy = true` κάνει το Terraform να αρνείται κάθε plan που θα το έσβηνε.



TERMINAL

```
mkdir ../guard
cd ../guard
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

main.tf

```
resource "local_file" "db" {
  filename = "customers.db"
  content  = "πελάτες\n"

  lifecycle {
    prevent_destroy = true
  }
}

resource "terraform_data" "app" {
  input = "app-1"
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Terraform planned the following actions, but then encountered a problem:
…
Plan: 0 to add, 0 to change, 2 to destroy.
╷
│ Error: Instance cannot be destroyed
│
│   on main.tf line 1:
│    1: resource "local_file" "db" {
│
│ Resource local_file.db has lifecycle.prevent_destroy set, but the plan
│ calls for this resource to be destroyed. To avoid this error and continue
│ with the plan, either disable lifecycle.prevent_destroy or reduce the scope
│ of the plan using the -target option.
╵
```

Το Terraform έφτιαξε το plan του destroy με `2 to destroy`. Μετά σταμάτησε με `Instance cannot be destroyed`. Το μήνυμα δείχνει το resource και λέει τις δύο διεξόδους. Αφαιρείς το `prevent_destroy` ή περιορίζεις το plan με `-target`, όπως στο κεφάλαιο 3. Ένα plan με σφάλμα δεν εκτελείται καθόλου, οπότε δεν σβήστηκε ούτε το `terraform_data.app`.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
local_file.db
terraform_data.app
```

Το ίδιο σφάλμα βγαίνει και σε αντικατάσταση, γιατί κάθε αντικατάσταση περιέχει ένα destroy. Άλλαξε το όνομα του αρχείου.

main.tf

```
resource "local_file" "db" {
  filename = "customers.sqlite"
  content  = "πελάτες\n"

  lifecycle {
    prevent_destroy = true
  }
}

resource "terraform_data" "app" {
  input = "app-1"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # local_file.db must be replaced
…
│ Error: Instance cannot be destroyed
…
```

Η προστασία όμως ζει μέσα στο block του resource. Αν σβήσεις όλο το block από τον κώδικα, μαζί του φεύγει και το `prevent_destroy`.

main.tf

```
resource "terraform_data" "app" {
  input = "app-1"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
  # local_file.db will be destroyed
  # (because local_file.db is not in configuration)
…
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.
```

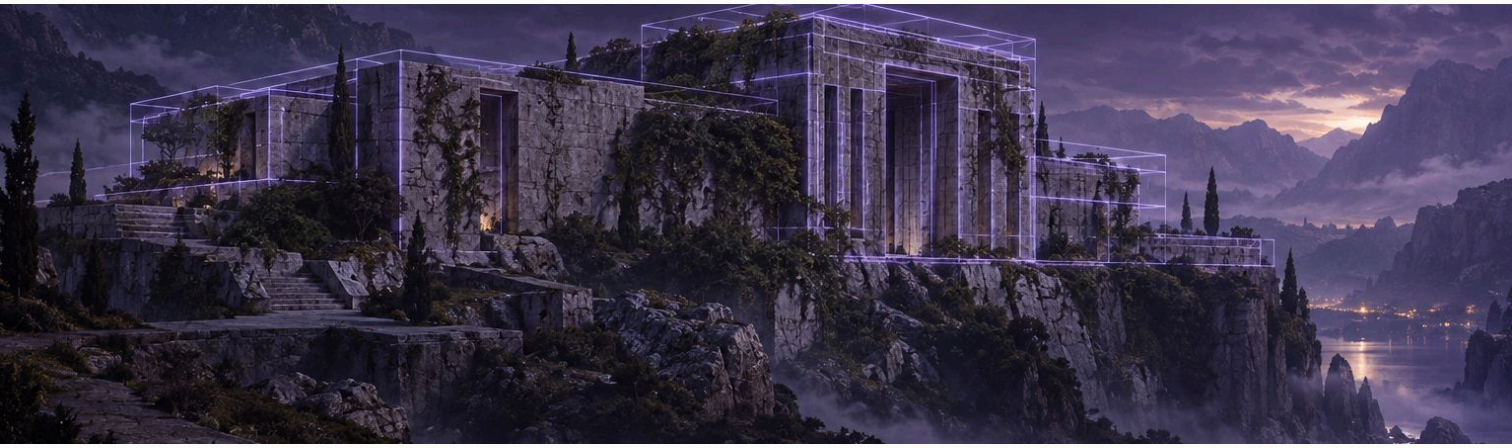
TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
terraform.tfstate.backup
versions.tf
```

Το Terraform βρήκε στο state ένα resource που δεν υπάρχει πια στη configuration. Το plan το λέει με τη γραμμή `because local_file.db is not in configuration`. Το έσβησε χωρίς καμία αντίρρηση. Το `customers.db` χάθηκε.



ΠΑΓΙΔΑ

Το `prevent_destroy` προστατεύει μόνο όσο το block του resource υπάρχει στον κώδικα. Δεν προστατεύει από διαγραφή του block, ούτε από διαγραφή μέσω της κονσόλας ή του CLI του cloud. Για σημαντικά δεδομένα συνδύασέ το με την προστασία του ίδιου του παρόχου, για παράδειγμα το `deletion_protection` μιας βάσης στο AWS.

ignore_changes

Μερικές φορές ένα attribute το αλλάζει κάποιος άλλος. Ένας autoscaler αλλάζει τον αριθμό των instances ανάλογα με τον φόρτο. Ένα εργαλείο της εταιρείας προσθέτει tags. Χωρίς ρύθμιση, κάθε plan θα έβλεπε τη διαφορά και θα γύριζε την τιμή πίσω σε αυτή του κώδικα. Το `ignore_changes` παίρνει μια λίστα από attributes. Το Terraform χρησιμοποιεί την τιμή τους μόνο στη δημιουργία και μετά δεν τα συγκρίνει πια.

Το παράδειγμα είναι μια υπηρεσία με όνομα και αριθμό αντιγράφων. Το όνομα το ορίζει ο κώδικας. Τα αντίγραφα τα ορίζει ένας autoscaler μετά τη δημιουργία. Το `ignore_changes` δέχεται και πεδία μέσα σε ένα object, εδώ το `input.replicas`.

TERMINAL

```
mkdir ../scale
cd ../scale
```

main.tf

```
resource "terraform_data" "service" {
  input = { name = "web", replicas = 2 }

  lifecycle {
    ignore_changes = [input.replicas]
  }
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

Ένα `terraform_data` δεν το αλλάζει κανείς έξω από το Terraform. Για να δεις τον μηχανισμό, άλλαξε την τιμή στον κώδικα. Για το Terraform η διαφορά ανάμεσα σε κώδικα και state είναι η ίδια, όποια πλευρά κι αν άλλαξε.

main.tf

```
resource "terraform_data" "service" {
  input = { name = "web", replicas = 5 }

  lifecycle {
    ignore_changes = [input.replicas]
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
terraform_data.service: Refreshing state... [id=936e90b6-66b3-bc59-cb8b-8af2edc89809]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το plan λέει `No changes`, παρότι ο κώδικας γράφει `5` και το state `2`. Άλλαξε τώρα και το όνομα.

main.tf

```
resource "terraform_data" "service" {
  input = { name = "api", replicas = 5 }

  lifecycle {
    ignore_changes = [input.replicas]
  }
}
```

TERMINAL

```
terraform apply -auto-approve
```

TERMINAL

```
terraform state show terraform_data.service
```

```
ΕΞΟΔΟΣ

# terraform_data.service:
resource "terraform_data" "service" {
    id     = "936e90b6-66b3-bc59-cb8b-8af2edc89809"
    input  = {
        name     = "api"
        replicas = 2
    }
    output = {
        name     = "api"
        replicas = 2
    }
}
```

Το όνομα έγινε `api`, ενώ τα `replicas` έμειναν `2`. Το `ignore_changes` αγνόησε μόνο το πεδίο που του είπες. Αν θέλεις να αγνοήσεις όλα τα attributes, γράφεις τη λέξη `all` χωρίς αγκύλες.

```
HCL

resource "terraform_data" "service" {
  input = { name = "api", replicas = 5 }

  lifecycle {
    ignore_changes = all
  }
}
```

Τρία σημεία θέλουν προσοχή. Το `ignore_changes` αγνοεί μόνο τη διαφορά στα attributes της λίστας. Αν το resource αντικατασταθεί για άλλο λόγο, το νέο αντικείμενο φτιάχνεται με τις τιμές του κώδικα. Γράφεις τα ονόματα χωρίς εισαγωγικά. Με εισαγωγικά το Terraform 1.x βγάζει την προειδοποίηση `Quoted references are deprecated`. Τέλος, ένα attribute που υπολογίζει μόνο ο provider, όπως το `output` του `terraform_data`, δεν έχει τιμή στον κώδικα για να αγνοηθεί. Αν το γράψεις, το Terraform σε προειδοποιεί με `Redundant ignore_changes element`. Η άσκηση 14.3 λύνει με το `ignore_changes` το πρόβλημα του `timestamp()` από το κεφάλαιο 10.

replace_triggered_by

Το `replace_triggered_by` κάνει το αντίθετο. Ζητά αντικατάσταση ενός resource όταν αλλάζει κάτι άλλο, ακόμη κι αν το ίδιο δεν άλλαξε καθόλου. Το κλασικό παράδειγμα είναι ένας server που πρέπει να ξαναστηθεί όταν αλλάζουν οι ρυθμίσεις του.

```
TERMINAL

mkdir ../restart
cd ../restart
```

main.tf

```
resource "terraform_data" "config" {
  input = { port = 8080, workers = 4 }
}

resource "terraform_data" "server" {
  input = "web-1"

  lifecycle {
    replace_triggered_by = [terraform_data.config]
  }
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

Άλλαξε τα `workers` από `4` σε `8`.

main.tf

```
resource "terraform_data" "config" {
  input = { port = 8080, workers = 8 }
}

resource "terraform_data" "server" {
  input = "web-1"

  lifecycle {
    replace_triggered_by = [terraform_data.config]
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # terraform_data.config will be updated in-place
…
  # terraform_data.server will be replaced due to changes in replace_triggered_by
…
Plan: 1 to add, 1 to change, 1 to destroy.
…
```

Το `config` αλλάζει επί τόπου. Ο `server` αντικαθίσταται και η επικεφαλίδα λέει γιατί, `due to changes in replace_triggered_by`. Μια αναφορά σε ολόκληρο resource ενεργοποιείται με κάθε αλλαγή του, update ή αντικατάσταση.

Μπορείς να γίνεις πιο συγκεκριμένος με αναφορά σε attribute. Τότε η αντικατάσταση γίνεται μόνο όταν αλλάζει η τιμή του. Κάνε apply, γράψε αναφορά στο `input.port` και άλλαξε ξανά μόνο τα `workers`.

TERMINAL

```
terraform apply -auto-approve
```

main.tf

```
resource "terraform_data" "config" {
  input = { port = 8080, workers = 16 }
}

resource "terraform_data" "server" {
  input = "web-1"

  lifecycle {
    replace_triggered_by = [terraform_data.config.input.port]
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # terraform_data.config will be updated in-place
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

Τώρα αλλάζει μόνο το `config`. Ο `server` θα αντικατασταθεί μόνο όταν αλλάξει το `port`.

Μόνο αναφορές σε resources

Το `replace_triggered_by` δέχεται μόνο αναφορές σε managed resources. Μια μεταβλητή δεν γίνεται δεκτή.

main.tf

```
variable "release" {
  type    = string
  default = "1.0"
}

resource "terraform_data" "server" {
  input = "web-1"

  lifecycle {
    replace_triggered_by = [var.release]
  }
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid reference in replace_triggered_by expression
╵
│ Only resources, count.index, and each.key may be used in
╷
│ Error: Invalid replace_triggered_by expression
╵
```

Το μήνυμα το λέει καθαρά. Στο `replace_triggered_by` επιτρέπονται μόνο resources, `count.index` και `each.key`. Τα δύο τελευταία, από το κεφάλαιο 11, χρησιμεύουν μόνο για να διαλέξεις instance, όπως `terraform_data.config[count.index]`. Η λύση για μια μεταβλητή είναι να την περάσεις από ένα `terraform_data`. Όταν αλλάζει η μεταβλητή, αλλάζει το `input` του. Αυτή η αλλαγή ενεργοποιεί το `replace_triggered_by`.

TERMINAL

```
mkdir ../release
cd ../release
```

main.tf

```
variable "release" {
  type    = string
  default = "1.0"
}

resource "terraform_data" "release" {
  input = var.release
}

resource "terraform_data" "server" {
  input = "web-1"

  lifecycle {
    replace_triggered_by = [terraform_data.release]
  }
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

TERMINAL

```
terraform plan -var release=1.1
```

ΕΞΟΔΟΣ

```
…
  # terraform_data.release will be updated in-place
…
  # terraform_data.server will be replaced due to changes in replace_triggered_by
…
Plan: 1 to add, 1 to change, 1 to destroy.
…
```

triggers_replace και keepers

Μερικά resources έχουν δικό τους μηχανισμό για το ίδιο πρόβλημα. Το `triggers_replace` του `terraform_data` το γνώρισες στο κεφάλαιο 6. Οι providers `random` και `time` έχουν αντίστοιχα arguments, τα `keepers` και `triggers`. Το `keepers` είναι ένα map. Όσο μένει ίδιο, η τυχαία τιμή μένει ίδια. Όταν αλλάξει, ο `random` βγάζει νέα τιμή.

TERMINAL

```
mkdir ../keepers
cd ../keepers
```

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

variable "image" {
  type    = string
  default = "ubuntu-24.04"
}

resource "random_pet" "server" {
  keepers = {
    image = var.image
  }
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

TERMINAL

```
terraform plan -var image=ubuntu-26.04
```

```
ΕΞΟΔΟΣ

…
  # random_pet.server must be replaced
…
      ~ keepers   = { # forces replacement
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Η αλλαγή στο `keepers` σημειώνεται με `# forces replacement`, όπως κάθε argument που δεν αλλάζει επί τόπου. Δεν είναι meta-argument. Είναι κανονικό argument του `random_pet` και δέχεται μεταβλητές.

Μηχανισμός	Πού γράφεται	Τι δέχεται
`replace_triggered_by`	Στο `lifecycle` κάθε resource	Μόνο αναφορές σε resources ή attributes τους
`triggers_replace`	Argument του `terraform_data`	Οποιαδήποτε τιμή, ακόμη και μεταβλητές
`keepers`	Argument των resources του `random`	Map από strings, ακόμη και από μεταβλητές
`triggers`	Argument του `time_sleep` και των άλλων resources του `time`	Map από strings, ακόμη και από μεταβλητές

Για resources άλλων providers, όπως ένα `aws_instance`, μόνο το `replace_triggered_by` υπάρχει. Αν η αιτία είναι μεταβλητή, την περνάς από `terraform_data`, όπως στο `release`.

Tainted resources

Ένα resource είναι **tainted** όταν το αντικείμενό του υπάρχει αλλά το Terraform δεν το εμπιστεύεται. Συμβαίνει όταν η δημιουργία ξεκίνησε, το αντικείμενο φτιάχτηκε, αλλά ένα επόμενο βήμα της δημιουργίας απέτυχε. Το Terraform κρατά το αντικείμενο στο state για να μη χαθεί, το σημαδεύει tainted και στο επόμενο plan το αντικαθιστά.

Για να δεις μια δημιουργία που αποτυγχάνει στη μέση, χρειάζεσαι ένα βήμα που τρέχει αφού φτιαχτεί το αντικείμενο. Αυτό είναι ένα **provisioner**, ένα nested block μέσα στο resource με τον τύπο του ως label. Εκτελείται μία φορά, αμέσως μετά τη δημιουργία. Το Terraform το θεωρεί μέρος της δημιουργίας. Αν αποτύχει, αποτυγχάνει και η δημιουργία, παρότι ο provider έχει ήδη φτιάξει το αντικείμενο. Ο τύπος `local-exec` τρέχει στον υπολογιστή σου την εντολή του argument `command`.

Η HashiCorp θεωρεί τα provisioners έσχατη λύση και το βιβλίο δεν τα χρησιμοποιεί αλλού. Εδώ χρησιμεύουν μόνο για να δεις ένα tainted resource. Η εντολή `test -f` ελέγχει αν υπάρχει το `app-1.0.zip`, το πακέτο της εφαρμογής. Δεν υπάρχει, οπότε η εντολή αποτυγχάνει.

```
TERMINAL

mkdir ../deploy
cd ../deploy
```

main.tf

```
resource "terraform_data" "deploy" {
  input = "app-1.0.zip"

  provisioner "local-exec" {
    command = "test -f app-1.0.zip"
  }
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
terraform_data.deploy: Creating...
terraform_data.deploy: Provisioning with 'local-exec'...
terraform_data.deploy (local-exec): Executing: ["/bin/sh" "-c" "test -f app-1.0.zip"]
…
│ Error: local-exec provisioner error
…
│ Error running command 'test -f app-1.0.zip': exit status 1. Output:
…
```

Η δημιουργία ξεκίνησε με `Creating...`. Μετά το `Provisioning with 'local-exec'` η εντολή απέτυχε με `exit status 1` και η γραμμή `Creation complete` δεν ήρθε ποτέ. Δες τι κράτησε το state.

TERMINAL

```
terraform state show terraform_data.deploy
```

ΕΞΟΔΟΣ

```
# terraform_data.deploy: (tainted)
resource "terraform_data" "deploy" {
    id     = "42d36186-cd2a-7b82-8b3e-dbf0d8281c1f"
    input  = "app-1.0.zip"
    output = "app-1.0.zip"
}
```

Το resource υπάρχει στο state με όλα του τα attributes και η πρώτη γραμμή γράφει `(tainted)`. Το επόμενο plan το αντικαθιστά.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # terraform_data.deploy is tainted, so must be replaced
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Η επικεφαλίδα γράφει `is tainted, so must be replaced`. Φτιάξε το πακέτο που έλειπε και κάνε apply. Το Terraform σβήνει το tainted αντικείμενο, φτιάχνει νέο και αυτή τη φορά το provisioner πετυχαίνει.

TERMINAL

```
touch app-1.0.zip
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
terraform_data.deploy: Refreshing state... [id=42d36186-cd2a-7b82-8b3e-dbf0d8281c1f]
…
terraform_data.deploy: Destroying... [id=42d36186-cd2a-7b82-8b3e-dbf0d8281c1f]
terraform_data.deploy: Destruction complete after 0s
terraform_data.deploy: Creating...
terraform_data.deploy: Provisioning with 'local-exec'...
terraform_data.deploy (local-exec): Executing: ["/bin/sh" "-c" "test -f app-1.0.zip"]
terraform_data.deploy: Creation complete after 0s [id=10517f8d-b2fb-d62e-a910-4526ae33823e]
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

ΣΗΜΕΙΩΣΗ

Μην μπερδεύεις το tainted με το `postcondition` του κεφαλαίου 13. Ένα `postcondition` που αποτυγχάνει μετά τη δημιουργία δεν κάνει το resource tainted. Το αντικείμενο μένει στο state κανονικά και το επόμενο plan σταματά στο ίδιο `postcondition` μέχρι να διορθώσεις την αιτία.

taint και untaint

Ένα resource γίνεται tainted και με το χέρι. Η εντολή `terraform taint` σημαδεύει ένα resource για αντικατάσταση στο επόμενο apply. Η `terraform untaint` σβήνει το σημάδι. Η δεύτερη χρησιμεύει όταν μια αποτυχημένη δημιουργία άφησε tainted ένα αντικείμενο που ξέρεις ότι είναι σωστό.

TERMINAL

```
terraform taint terraform_data.deploy
terraform untaint terraform_data.deploy
```

ΕΞΟΔΟΣ

```
Resource instance terraform_data.deploy has been marked as tainted.
Resource instance terraform_data.deploy has been successfully untainted.
```

Και οι δύο εντολές γράφουν στο state αμέσως, χωρίς plan και χωρίς ερώτηση. Αυτό είναι το πρόβλημα του `taint`. Σε μια ομάδα με κοινό state, το επόμενο plan οποιουδήποτε συναδέλφου θα περιέχει μια αντικατάσταση που δεν ζήτησε. Η τεκμηρίωση της HashiCorp χαρακτηρίζει το `taint` deprecated. Προτείνει το `-replace` του κεφαλαίου 3, που υπάρχει από το Terraform 0.15.2. Το `-replace` δεν γράφει

τίποτα στο state από πριν. Εμφανίζει την αντικατάσταση μέσα στο plan, που μπορείς να διαβάσεις πριν από το apply.

TERMINAL

```
terraform plan -replace=terraform_data.deploy
```

ΕΞΟΔΟΣ

```
…
  # terraform_data.deploy will be replaced, as requested
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Για αντικατάσταση χωρίς αλλαγή στον κώδικα η απάντηση είναι `terraform apply -replace=ADDRESS`. Το `terraform taint` υπάρχει ακόμη αλλά δεν προτείνεται. Ένα resource γίνεται tainted και μόνο του, όταν η δημιουργία του αποτύχει αφού το αντικείμενο έχει ήδη φτιαχτεί. Στο επόμενο apply αντικαθίσταται.

Το destroy = false του 1.16

ΣΗΜΕΙΩΣΗ

Το Terraform 1.16 πρόσθεσε στο `lifecycle` το argument `destroy`. Είναι εκτός ύλης της εξέτασης. Με `destroy = false` το Terraform δεν σβήνει ποτέ το αντικείμενο. Σε `terraform destroy` το αφαιρεί μόνο από το state και το plan γράφει `will no longer be managed by Terraform, but will not be destroyed`. Σε αντικατάσταση αφήνει το παλιό αντικείμενο να υπάρχει έξω από το Terraform και φτιάχνει νέο. Μοιάζει με το block `removed` με `destroy = false`, που γνωρίζεις στο κεφάλαιο 23. Εκείνο ισχύει από την 1.7 και είναι στην ύλη.

Τα arguments του lifecycle σε μία σελίδα

Argument	Τι κάνει	Πότε το χρησιμοποιείς
`create_before_destroy`	Create πριν από destroy, σύμβολο `+/-`	Όταν η αντικατάσταση δεν πρέπει να αφήσει κενό και το όνομα δεν συγκρούεται
`prevent_destroy`	Σφάλμα σε κάθε plan που σβήνει το resource	Για βάσεις και δεδομένα που δεν ξαναφτιάχνονται
`ignore_changes`	Αγνοεί αλλαγές σε attributes μετά τη δημιουργία	Όταν ένα attribute το αλλάζει άλλος μηχανισμός
`replace_triggered_by`	Αντικατάσταση όταν αλλάζει άλλο resource ή attribute	Όταν ένα resource πρέπει να ξαναστηθεί μαζί με άλλο
`precondition`	Έλεγχος πριν από την αλλαγή, κεφάλαιο 13	Για προϋποθέσεις που πρέπει να ισχύουν
`postcondition`	Έλεγχος μετά την αλλαγή, κεφάλαιο 13	Για εγγυήσεις για το αποτέλεσμα

Argument	Τι κάνει	Πότε το χρησιμοποιείς
`action_trigger`	Εκτελεί ένα action πριν ή μετά από create ή update, Terraform 1.14	Εκτός ύλης
`destroy`	Με `false` αφαιρεί από το state αντί να σβήσει, Terraform 1.16	Εκτός ύλης

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch14`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από υπάρχοντα. Χρησιμοποίησε `-auto-approve` όπου θέλεις.

ΑΣΚΗΣΗ 14.1 Μοναδικό όνομα

Ξεκίνα από αντίγραφο του φακέλου `names`, χωρίς state. Άλλαξε το `filename` ώστε να περιέχει την έκδοση, για παράδειγμα `release-1.0.html`. Κράτα το `create_before_destroy`. Κάνε apply με την προεπιλεγμένη έκδοση και μετά με `-var release=1.1`. Το `ls` στο τέλος πρέπει να δείχνει μόνο τη νέα σελίδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
main.tf
release-1.1.html
terraform.tfstate
terraform.tfstate.backup
versions.tf
```

ΑΣΚΗΣΗ 14.2 Σβήσε μόνο ό,τι επιτρέπεται

Γράψε configuration με δύο `terraform_data`. Το `db` έχει `input = "customers"` και `prevent_destroy`. Το `cache` έχει `input = "redis"`. Κάνε apply. Σβήσε μετά μόνο το `cache` χωρίς να αλλάξεις τον κώδικα. Δείξε τι έμεινε στο state.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
terraform_data.db
```

ΑΣΚΗΣΗ 14.3 Η ώρα δημιουργίας

Στο κεφάλαιο 10 ένα `local_file` με `timestamp()` στο `content` αντικαθιστούνταν σε κάθε plan. Γράψε το `local_file.stamp` με `filename = "stamp.txt"` και `content = "Δημιουργήθηκε ${timestamp()}\n"` ώστε να κρατά την ώρα της πρώτης δημιουργίας. Κάνε apply και μετά plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
No changes. Your infrastructure matches the configuration.
…
```

ΑΣΚΗΣΗ 14.4 Restart όταν αλλάζει η έκδοση

Γράψε configuration με μια μεταβλητή `app_version` με default `"1.0"` και ένα `local_file.app` που γράφει το `app.conf` με περιεχόμενο `port = 8080\n`. Το αρχείο δεν περιέχει την έκδοση, αλλά πρέπει να ξαναφτιάχνεται κάθε φορά που αλλάζει η `app_version`. Αν χρειαστείς βοηθητικό resource, δώσε του το όνομα `app_version`. Κάνε apply και μετά plan με `-var app_version=1.1`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # local_file.app will be replaced due to changes in replace_triggered_by
…
  # terraform_data.app_version will be updated in-place
…
Plan: 1 to add, 1 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 14.5 keepers ή όχι

Γράψε configuration με δύο μεταβλητές, `image` με default `"ubuntu-24.04"` και `size` με default `"small"`. Ένα `random_pet.name` πρέπει να αλλάζει μόνο όταν αλλάζει το `image`. Ένα `local_file.server` γράφει το `server.txt` με το όνομα και το `size`. Κάνε apply. Δείξε ότι ένα plan με `-var size=large` δεν αλλάζει το όνομα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # local_file.server must be replaced
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 14.6 Αντικατάσταση παρά το ignore_changes

Γράψε δύο `terraform_data`. Το `config` έχει `input = "v1"`. Το `server` έχει `input = "owner-anna"`, αγνοεί τις αλλαγές στο `input` και αντικαθίσταται όταν αλλάζει το `config`. Κάνε apply. Άλλαξε το `input` του `server` σε `"owner-nikos"` και επιβεβαίωσε με plan ότι δεν αλλάζει τίποτα. Άλλαξε μετά το `input` του `config` σε `"v2"` και κάνε apply. Ποια τιμή έχει τώρα το `input` του `server` στο state;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
# terraform_data.server:
resource "terraform_data" "server" {
    id     = "5652ceff-e869-5728-25ab-78e9518940f2"
    input  = "owner-nikos"
    output = "owner-nikos"
}
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 14.1

Κάθε αλλαγή στο image ενός server αντικαθιστά τον server και η υπηρεσία μένει εκτός για λίγα λεπτά. Ποια ρύθμιση κάνει το Terraform να φτιάχνει τον νέο server πριν σβήσει τον παλιό;

a. `prevent_destroy = true`
b. `create_before_destroy = true`
c. `ignore_changes = [ami]`
d. `replace_triggered_by = [aws_instance.web]`

ΕΡΩΤΗΣΗ 14.2

Ένα resource έχει `prevent_destroy = true`. Ποιες προτάσεις ισχύουν; Επίλεξε δύο.

a. Το `terraform destroy` αποτυγχάνει και δεν σβήνει κανένα resource.
b. Μια αλλαγή που απαιτεί αντικατάσταση εκτελείται κανονικά.
c. Αν σβήσεις το block του resource από τον κώδικα, το επόμενο apply το σβήνει.
d. Το resource προστατεύεται και από διαγραφή μέσω της κονσόλας του cloud.
e. Η τιμή μπορεί να έρθει από μεταβλητή, για παράδειγμα `var.protect`.

ΕΡΩΤΗΣΗ 14.3

Ένας autoscaler αλλάζει το `desired_capacity` ενός `aws_autoscaling_group` ανάλογα με τον φόρτο. Κάθε `terraform apply` το επαναφέρει στην τιμή του κώδικα. Τι πρέπει να κάνεις;

a. Να προσθέσεις `prevent_destroy = true`.
b. Να προσθέσεις `create_before_destroy = true`.
c. Να προσθέσεις `ignore_changes = [desired_capacity]` στο `lifecycle`.
d. Να τρέχεις `terraform apply -replace=aws_autoscaling_group.web`.

ΕΡΩΤΗΣΗ 14.4

Γράφεις `replace_triggered_by = [var.app_version]` σε ένα `aws_instance`. Τι συμβαίνει;

a. Το instance αντικαθίσταται κάθε φορά που αλλάζει η `app_version`.
b. Το Terraform βγάζει σφάλμα, γιατί δέχεται μόνο αναφορές σε resources.
c. Η ρύθμιση αγνοείται σιωπηλά.
d. Το instance αντικαθίσταται σε κάθε apply.

ΕΡΩΤΗΣΗ 14.5

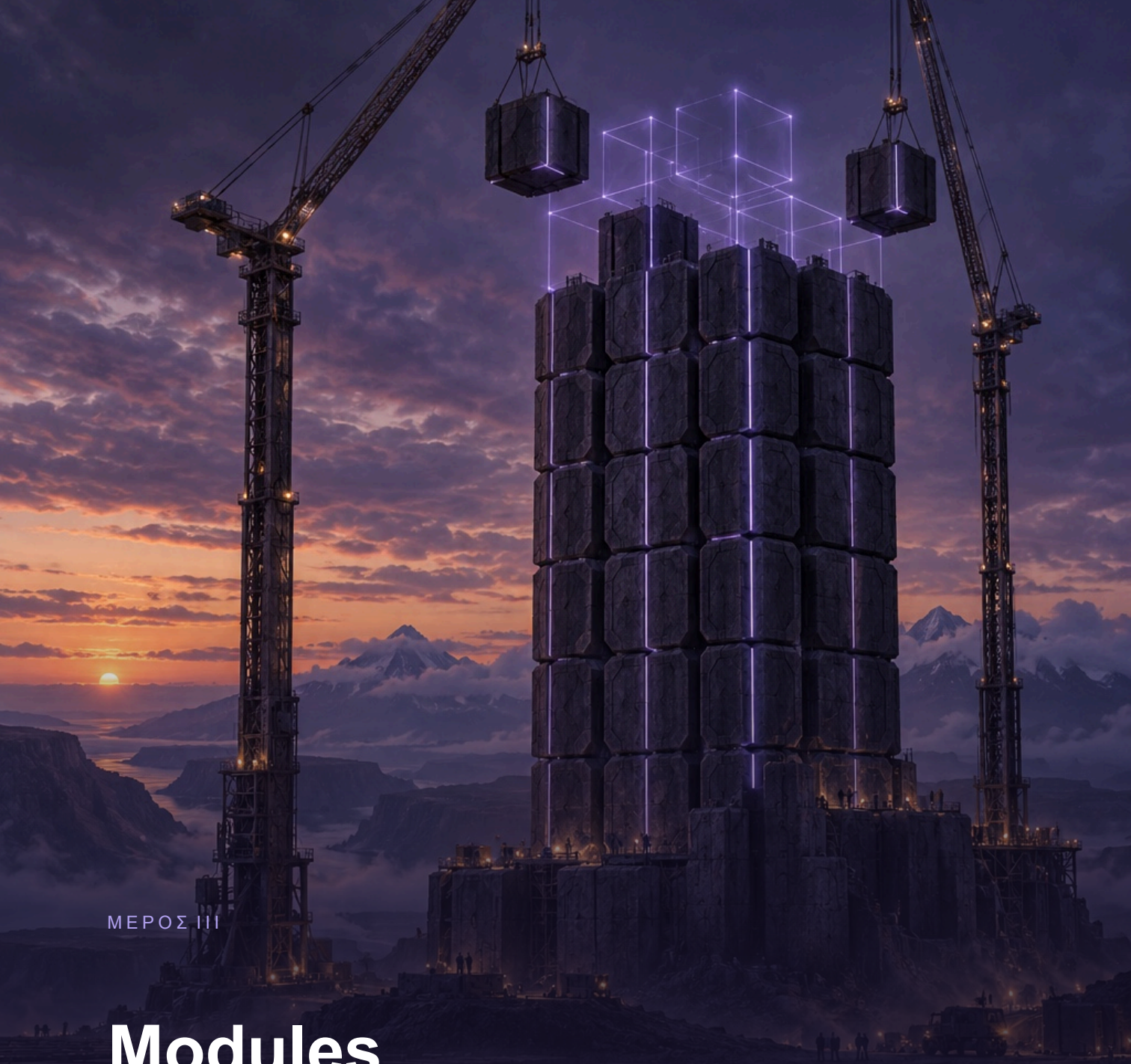
Κατά το `terraform apply` ο provider φτιάχνει ένα instance, αλλά το provisioner του αποτυγχάνει και το apply τελειώνει με σφάλμα. Τι κάνει το επόμενο `terraform apply`, αν δεν αλλάξεις τίποτα;

a. Τίποτα, γιατί το instance υπάρχει ήδη στο state.
b. Φτιάχνει δεύτερο instance και κρατά και το πρώτο.
c. Αποτυγχάνει μέχρι να τρέξεις `terraform untaint`.
d. Αντικαθιστά το instance, γιατί είναι σημαδεμένο ως tainted.

ΕΡΩΤΗΣΗ 14.6

Σωστό ή λάθος; Αν ένα resource έχει `create_before_destroy = true`, τα resources από τα οποία εξαρτάται αντικαθίστανται κι αυτά με create πριν από destroy, ακόμη κι αν δεν έχουν δικό τους `lifecycle`.

a. Σωστό
b. Λάθος



ΜΕΡΟΣ ΙΙΙ

Modules

Μέχρι εδώ κάθε configuration ήταν ένας φάκελος που τα έκανε όλα μόνος του. Σε αυτό το μέρος κόβεις τον κώδικα σε κομμάτια με όνομα, είσοδο και έξοδο, που τα ξαναχρησιμοποιείς όσες φορές θέλεις. Πρώτα γράφεις τα δικά σου modules. Μετά φέρνεις έτοιμα από το Registry και από Git.



ΚΕΦΑΛΑΙΟ 15

Το πρώτο module

Ένα module είναι ένας φάκελος με αρχεία `.tf` που τον καλείς από άλλη configuration, όπως καλείς μια συνάρτηση. Σε αυτό το κεφάλαιο γράφεις ένα module που φτιάχνει μια μικρή στατική ιστοσελίδα, το καλείς μία, δύο και τρεις φορές και μαθαίνεις ποιες τιμές περνούν τα όριά του και ποιες όχι.

Τι είναι module

Κάθε φάκελος με αρχεία `.tf` είναι **module**. Δεν χρειάζεται δήλωση ούτε ειδικό αρχείο. Όλες οι configurations που έγραψες ως τώρα ήταν modules χωρίς να το ξέρεις.

Ο φάκελος όπου τρέχεις `terraform plan` και `apply` είναι το **root module**, όπως είπε το κεφάλαιο 1. Ένα module μπορεί να καλεί άλλα modules με το block `module`. Το module που καλείται λέγεται **child module** και αυτό που το καλεί λέγεται parent ή calling module. Ένα child module μπορεί να καλεί κι αυτό άλλα, οπότε τα modules μιας configuration σχηματίζουν δέντρο με το root στην κορυφή.

Το child module μοιάζει με συνάρτηση. Οι input variables του κεφαλαίου 7 είναι οι παράμετροί της και τα outputs του κεφαλαίου 8 οι τιμές που επιστρέφει. Τα resources, τα data sources και τα locals του μένουν κρυμμένα από όποιον το καλεί.

Τρεις λόγοι κάνουν τα modules τόσο κοινά.

- **Επαναχρησιμοποίηση.** Γράφεις μία φορά πώς στήνεται κάτι και το καλείς όσες φορές χρειάζεσαι, με διαφορετικές τιμές. Το ίδιο module δικτύου φτιάχνει το δίκτυο του development και του production.
- **Ενθυλάκωση.** Όποιος καλεί το module βλέπει λίγες μεταβλητές και λίγα outputs, όχι τα δεκάδες resources που υπάρχουν από πίσω. Το χρησιμοποιεί χωρίς να ξέρει κάθε λεπτομέρεια.
- **Συνέπεια.** Όλες οι κλήσεις ακολουθούν τους ίδιους κανόνες, για παράδειγμα τα ίδια ονόματα ή τις ίδιες ρυθμίσεις ασφαλείας. Μια διόρθωση στο module φτάνει σε κάθε κλήση με το επόμενο apply.

Ένα module για στατική ιστοσελίδα

Το module του κεφαλαίου γράφει τα αρχεία μιας μικρής στατικής ιστοσελίδας, μια σελίδα HTML και ένα stylesheet. Φτιάξε τον φάκελο του παραδείγματος.

TERMINAL

```
mkdir web
cd web
```

Το module ζει στον υποφάκελο `modules/site`. Το όνομα `modules` είναι σύμβαση, όχι κανόνας. Τα αρχεία του χωρίζονται ανά ρόλο, όπως στο κεφάλαιο 8. Το `versions.tf` λέει ποιους providers χρειάζεται.

modules/site/versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = ">= 2.9.1"
    }
  }
}
```

Το constraint είναι `>=` και όχι ακριβής έκδοση. Το κεφάλαιο 5 εξήγησε γιατί. Ένα module που θα το καλούν πολλές configurations δηλώνει μόνο την ελάχιστη έκδοση που χρειάζεται. Την ακριβή την καρφώνει το root module.

Το `variables.tf` είναι η είσοδος του module.

modules/site/variables.tf

```
variable "title" {
  type        = string
  description = "Ο τίτλος της ιστοσελίδας"
}

variable "output_dir" {
  type        = string
  description = "Ο φάκελος όπου γράφονται τα αρχεία"
}

variable "color" {
  type        = string
  default     = "navy"
  description = "Το χρώμα των επικεφαλίδων"
}
```

Οι `title` και `output_dir` δεν έχουν `default`, άρα είναι υποχρεωτικές. Κάθε κλήση πρέπει να τους δώσει τιμή. Η `color` έχει προεπιλογή, οπότε η κλήση τη γράφει μόνο αν θέλει άλλο χρώμα. Κάθε μεταβλητή έχει `description`. Σε ένα module αυτό είναι η τεκμηρίωση για όποιον το καλεί.

Το `main.tf` έχει τα resources και το `index.html.tftpl` είναι το template της σελίδας.

modules/site/main.tf

```
resource "local_file" "index" {
  filename = "${var.output_dir}/index.html"
  content = templatefile("${path.module}/index.html.tftpl", {
    title = var.title
  })
}

resource "local_file" "css" {
  filename = "${var.output_dir}/style.css"
  content  = "h1 { color: ${var.color}; }\n"
}
```

modules/site/index.html.tftpl

```
<!doctype html>
<html>
<head>
  <title>${title}</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>${title}</h1>
</body>
</html>
```

Μέσα στο module το `var.title` είναι η μεταβλητή `title` του module. Η `templatefile` του κεφαλαίου 10 βάζει την τιμή της στη θέση του `${title}`. Το path του template ξεκινά με `path.module`. Εδώ το `path.module` είναι `modules/site`, ο φάκελος του module, οπότε το module βρίσκει το template του όποιος κι αν το καλεί.

Το `filename` δεν χρησιμοποιεί `path.module`. Ένα σχετικό path στο `filename` μετράει από τον φάκελο όπου τρέχει το Terraform, δηλαδή από το root module. Πού θα γραφτούν τα αρχεία το αποφασίζει όποιος καλεί το module, με την `output_dir`. Ο κανόνας του κεφαλαίου 10 ισχύει για τα αρχεία που ανήκουν στο module, όπως το template.

Το `outputs.tf` είναι η έξοδος.

modules/site/outputs.tf

```
output "index_path" {
  value       = local_file.index.filename
  description = "Το path της αρχικής σελίδας"
}

output "files" {
  value       = [local_file.index.filename, local_file.css.filename]
  description = "Όλα τα αρχεία της ιστοσελίδας"
}
```

Μόνο αυτά τα δύο outputs θα βλέπει όποιος καλεί το module. Το module δεν έχει block `provider` και δεν τρέχεις ποτέ `plan` μέσα στον φάκελό του. Περιμένει ένα root module να το καλέσει και να του δώσει τιμές.

Η κλήση από το root module

Γράψε το `main.tf` του root module στον φάκελο `web`.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

module "site" {
  source = "./modules/site"

  title      = "Το blog μου"
  output_dir = "public/blog"
}

output "blog_index" {
  value = module.site.index_path
}
```

Διάβασε το block `module` γραμμή προς γραμμή.

- Το label `site` είναι το όνομα της κλήσης μέσα στο root module. Δεν είναι υποχρεωτικό να ταιριάζει με τον φάκελο. Θα μπορούσες να το λες `blog`.
- Το `source` λέει πού βρίσκεται ο κώδικας του module και είναι υποχρεωτικό. Ένα τοπικό path ξεκινά με `./` ή `../` και μετράει από τον φάκελο του module που κάνει την κλήση. Το κεφάλαιο 16 δείχνει πηγές από το Registry και από Git.
- Τα `title` και `output_dir` έχουν τα ονόματα των μεταβλητών του child module. Κάθε argument δίνει τιμή στη μεταβλητή με το ίδιο όνομα. Η `color` δεν δίνεται, οπότε παίρνει το `default`.

Το output `blog_index` διαβάζει το output `index_path` του child module με την έκφραση `module.site.index_path`. Το root module δηλώνει τον `local` με ακριβή έκδοση, αν και δεν έχει δικό του `local_file`. Έτσι καρφώνει την έκδοση που θα χρησιμοποιήσει και το child module.

Το init και τα modules

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
- site in modules/site

Initializing provider plugins...
- Finding hashicorp/local versions matching ">= 2.9.1, 2.9.1"...
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
```

Η έξοδος έχει νέα ενότητα, `Initializing modules...`, ανάμεσα στο backend και στους providers. Η γραμμή `- site in modules/site` λέει ότι η κλήση `site` βρέθηκε στον φάκελο `modules/site`. Στους providers η γραμμή `Finding` γράφει δύο constraints μαζί. Το `>= 2.9.1` είναι του child module και το `2.9.1` του root. Το Terraform διαλέγει μία έκδοση που ικανοποιεί τα constraints όλων των modules, εδώ την 2.9.1.

Το init σημειώνει τα modules που βρήκε στο `.terraform/modules/modules.json`.

TERMINAL

```
jq . .terraform/modules/modules.json
```

ΕΞΟΔΟΣ

```
{
  "Modules": [
    {
      "Key": "",
      "Source": "",
      "Dir": "."
    },
    {
      "Key": "site",
      "Source": "./modules/site",
      "Dir": "modules/site"
    }
  ]
}
```

Κάθε εγγραφή είναι ένα module του δέντρου. Η πρώτη, με κενό `Key`, είναι το root module στον φάκελο `.`. Η δεύτερη είναι η κλήση `site`. Το `Source` είναι ό,τι έγραψες στο `source` και το `Dir` ο φάκελος από όπου το Terraform διαβάζει τον κώδικα. Για τοπικό path το `Dir` είναι ο ίδιος ο φάκελος του module. Το init δεν αντιγράφει τίποτα. Γι' αυτό μια αλλαγή μέσα στο `modules/site` φαίνεται αμέσως στο επόμενο plan χωρίς νέο init. Ένα module από το Registry ή από Git κατεβαίνει σε υποφάκελο του `.terraform/modules`, όπως θα δεις στο κεφάλαιο 16.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.site.local_file.css will be created
  + resource "local_file" "css" {
      + content              = <<-EOT
            h1 { color: navy; }
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "public/blog/style.css"
      + id                   = (known after apply)
    }

  # module.site.local_file.index will be created
  + resource "local_file" "index" {
      + content              = <<-EOT
            <!doctype html>
            <html>
            <head>
              <title>Το blog μου</title>
              <link rel="stylesheet" href="style.css">
            </head>
            <body>
              <h1>Το blog μου</h1>
            </body>
            </html>
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "public/blog/index.html"
      + id                   = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + blog_index = "public/blog/index.html"

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Οι διευθύνσεις των resources έχουν πρόθεμα. Το plan γράφει `module.site.local_file.css` και `module.site.local_file.index`. Το `module.site` λέει σε ποια κλήση ανήκει το resource. Το `content` του `index` είναι το template με τον τίτλο στη θέση του `${title}`. Το stylesheet έχει το `navy`, την προεπιλογή της `color`. Στο `Changes to Outputs` το `blog_index` έχει ήδη τιμή, γιατί το `filename` βγαίνει μόνο από μεταβλητές.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
module.site.local_file.index: Creating...
module.site.local_file.css: Creating...
module.site.local_file.css: Creation complete after 0s [id=a04bee93254a768c54469c13ff7498949e40de2c]
module.site.local_file.index: Creation complete after 0s [id=283048654f43324eef2fbaeacae5f58416bbaf60]

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

Outputs:

blog_index = "public/blog/index.html"
```

TERMINAL

```
find public -type f
cat public/blog/index.html
```

ΕΞΟΔΟΣ

```
public/blog/index.html
public/blog/style.css
<!doctype html>
<html>
<head>
  <title>Το blog μου</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Το blog μου</h1>
</body>
</html>
```

Τα αρχεία γράφτηκαν στο `public/blog` μέσα στον φάκελο `web`, εκεί όπου έτρεξε το Terraform. Μέσα στο `modules/site` δεν γράφτηκε τίποτα.

ΠΑΓΙΔΑ

Το `source = "modules/site"` χωρίς `./` δεν είναι τοπικό path. Το Terraform το διαβάζει σαν διεύθυνση από το Registry και το init σταματά με `Invalid module source address`. Το μήνυμα προτείνει το `"./modules/site"`. Ένα τοπικό path ξεκινά πάντα με `./` ή `../`.

Δεύτερη κλήση και νέο init

Η αξία του module φαίνεται στη δεύτερη κλήση. Πρόσθεσε στο τέλος του `main.tf` μια ιστοσελίδα τεκμηρίωσης. Παράλειψε επίτηδες τον φάκελο.

main.tf

```
module "docs" {
  source = "./modules/site"

  title = "Τεκμηρίωση"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Module not installed
│
│   on main.tf line 20:
│   20: module "docs" {
│
│ This module is not yet installed. Run "terraform init" to install all
│ modules required by this configuration.
╵
```

Το Terraform σταματά με `Module not installed`. Διαβάζει τα modules από το `modules.json` και εκεί δεν υπάρχει ακόμη το `docs`. Κάθε νέο block `module`, ή αλλαγή στο `source` ενός υπάρχοντος, θέλει πρώτα εγκατάσταση. Ο ένας τρόπος είναι το `terraform init`. Ο άλλος είναι το `terraform get`, που εγκαθιστά μόνο τα modules, χωρίς backend και providers.

TERMINAL

```
terraform get
```

ΕΞΟΔΟΣ

```
- docs in modules/site
```

Το `get` τύπωσε μόνο τη νέα κλήση, γιατί το `site` ήταν ήδη εγκατεστημένο. Με `terraform get -update` ελέγχει ξανά και όσα υπάρχουν, κάτι που έχει νόημα για modules που κατεβαίνουν από αλλού. Στην πράξη τρέχεις συνήθως `terraform init`, που κάνει και τα δύο.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Missing required argument
│
│   on main.tf line 20, in module "docs":
│   20: module "docs" {
│
│ The argument "output_dir" is required, but no definition was found.
╵
```

Τώρα το σφάλμα είναι `Missing required argument`. Η `output_dir` δεν έχει `default` και η κλήση `docs` δεν της έδωσε τιμή. Πρόσεξε τη διαφορά από το κεφάλαιο 7. Μια υποχρεωτική μεταβλητή του root module που μένει χωρίς τιμή σε κάνει να απαντήσεις σε ερώτηση. Για μια μεταβλητή child module το Terraform δεν ρωτά ποτέ. Η τιμή γράφεται στο block `module`, οπότε η παράλειψη είναι λάθος της configuration και την πιάνει ήδη το `terraform validate`. Το ίδιο ισχύει για ένα argument που δεν

αντιστοιχεί σε μεταβλητή. Ένα `colour = "teal"` δίνει `Unsupported argument` και το μήνυμα ρωτά `Did you mean "color"?`.

Γράψε το `main.tf` ολόκληρο, με τις δύο κλήσεις και ένα output για καθεμία.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

module "site" {
  source = "./modules/site"

  title      = "Το blog μου"
  output_dir = "public/blog"
}

module "docs" {
  source = "./modules/site"

  title      = "Τεκμηρίωση"
  output_dir = "public/docs"
  color      = "teal"
}

output "blog_index" {
  value = module.site.index_path
}

output "docs_index" {
  value = module.docs.index_path
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
module.site.local_file.index: Refreshing state... [id=283048654f43324eef2fbaeacae5f58416bbaf60]
module.site.local_file.css: Refreshing state... [id=a04bee93254a768c54469c13ff7498949e40de2c]
…
Plan: 2 to add, 0 to change, 0 to destroy.
…
module.docs.local_file.index: Creating...
module.docs.local_file.css: Creating...
module.docs.local_file.css: Creation complete after 0s [id=0d6486781fa3970a1ca242653deae5e790508e31]
module.docs.local_file.index: Creation complete after 0s [id=acd5f76d084076826245c8867cd90dad98daea24]
…
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
…
```



TERMINAL

```
cat public/docs/style.css
```

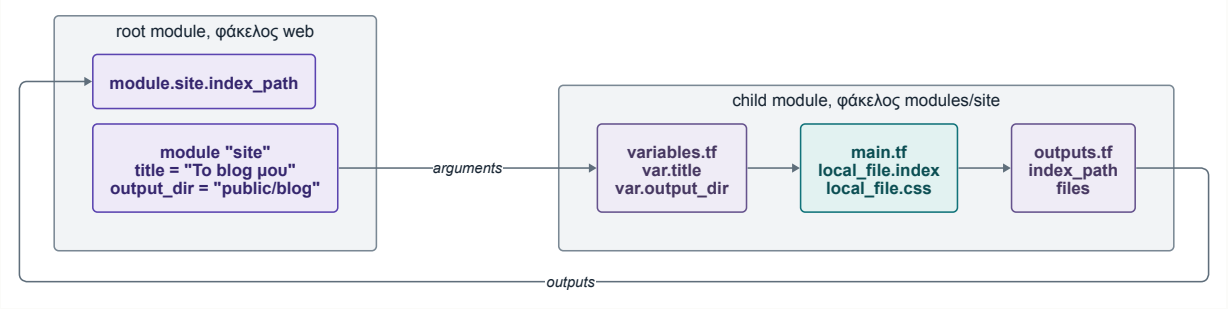
ΕΞΟΔΟΣ

```
h1 { color: teal; }
```

Το `site` δεν άλλαξε, οπότε το plan πρόσθεσε μόνο τα δύο αρχεία του `docs`. Ο ίδιος κώδικας έδωσε δεύτερη ιστοσελίδα με άλλον τίτλο, άλλο φάκελο και χρώμα `teal`.

Τι βλέπει κάθε module

Κάθε module έχει τον δικό του χώρο ονομάτων, το **scope** του. Μέσα στο `modules/site` το `var.title` σημαίνει τη μεταβλητή `title` του module. Οι μεταβλητές, τα locals και τα resources του root δεν υπάρχουν εκεί. Το ίδιο ισχύει ανάποδα.



Σχήμα 15.1 · Τα όρια ενός module. Οι τιμές μπαίνουν μόνο από τα arguments του block module στις variables του child και βγαίνουν μόνο από τα outputs του. Τα resources μένουν μέσα.

Δοκίμασε πρώτα από μέσα προς τα έξω. Δώσε στο root μια μεταβλητή και ένα local.

scope.tf

```
variable "owner" {
  type    = string
  default = "Μαρία"
}

locals {
  year = 2026
}
```

Γράψε στο module ένα resource που τα χρησιμοποιεί.

modules/site/footer.tf

```
resource "local_file" "footer" {
  filename = "${var.output_dir}/footer.html"
  content  = "<p>© ${local.year} ${var.owner}</p>\n"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Reference to undeclared local value
│
│   on modules/site/footer.tf line 3, in resource "local_file" "footer":
│    3:   content  = "<p>© ${local.year} ${var.owner}</p>\n"
│
│ A local value with the name "year" has not been declared.
╵

╷
│ Error: Reference to undeclared local value
│
│   on modules/site/footer.tf line 3, in resource "local_file" "footer":
│    3:   content  = "<p>© ${local.year} ${var.owner}</p>\n"
│
│ A local value with the name "year" has not been declared.
╵

╷
│ Error: Reference to undeclared input variable
│
│   on modules/site/footer.tf line 3, in resource "local_file" "footer":
│    3:   content  = "<p>© ${local.year} ${var.owner}</p>\n"
│
│ An input variable with the name "owner" has not been declared. This
│ variable can be declared with a variable "owner" {} block.
╵

╷
│ Error: Reference to undeclared input variable
│
│   on modules/site/footer.tf line 3, in resource "local_file" "footer":
│    3:   content  = "<p>© ${local.year} ${var.owner}</p>\n"
│
│ An input variable with the name "owner" has not been declared. This
│ variable can be declared with a variable "owner" {} block.
╵
```

Το validate βγάζει τέσσερα σφάλματα, όλα στο `modules/site/footer.tf`. Είναι δύο σφάλματα που εμφανίζονται δύο φορές, μία για κάθε κλήση του module, την `site` και την `docs`. Το local `year` και η μεταβλητή `owner` δεν έχουν δηλωθεί. Το root τα έχει, αλλά για το child δεν υπάρχουν. Ένα child module δεν κληρονομεί μεταβλητές, locals ή resources από το parent. Αν χρειάζεται μια τιμή, τη δηλώνει ως δική του μεταβλητή και το parent τη δίνει ρητά στην κλήση.

HCL

```
module "site" {
  source = "./modules/site"

  title      = "Το blog μου"
  output_dir = "public/blog"
  owner      = var.owner
}
```

Αριστερά του `=` είναι η μεταβλητή `owner` του child. Δεξιά είναι η `var.owner` του root, που εκεί υπάρχει. Τα δύο ονόματα μπορούν να διαφέρουν. Σβήσε τα δύο αρχεία της δοκιμής.

TERMINAL

```
rm scope.tf modules/site/footer.tf
```

Τώρα από έξω προς τα μέσα. Ζήτησε από το root ένα output που το module δεν έχει και ένα resource του module.

broken.tf

```
output "docs_url" {
  value = module.docs.url
}

output "docs_css" {
  value = module.docs.local_file.css.filename
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Unsupported attribute
│
│   on broken.tf line 2, in output "docs_url":
│    2:   value = module.docs.url
│     ├────────────────
│     │ module.docs is object with 2 attributes
│
│ This object does not have an attribute named "url".
╵

╷
│ Error: Unsupported attribute
│
│   on broken.tf line 6, in output "docs_css":
│    6:   value = module.docs.local_file.css.filename
│     ├────────────────
│     │ module.docs is object with 2 attributes
│
│ This object does not have an attribute named "local_file".
╵
```

Και τα δύο σφάλματα είναι `Unsupported attribute`. Η γραμμή `module.docs is object with 2 attributes` εξηγεί το γιατί. Για το root, το `module.docs` είναι ένα object με ένα attribute για κάθε output του child, το `index_path` και το `files`. Δεν έχει `url`, γιατί τέτοιο output δεν υπάρχει. Δεν έχει ούτε `local_file`, γιατί τα resources του child δεν βγαίνουν ποτέ έξω. Αν το root χρειάζεται το path του stylesheet, το module πρέπει να το δώσει με output.

TERMINAL

```
rm broken.tf
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ελέγχει το scope με δύο τρόπους. Ένα child module δεν βλέπει τις μεταβλητές, τα locals και τα resources του parent. Παίρνει τιμές μόνο από τα arguments του block `module`. Το parent δεν βλέπει τα resources του child. Διαβάζει μόνο τα outputs του, με τη μορφή `module.ΟΝΟΜΑ.OUTPUT`, όπου ΟΝΟΜΑ είναι το label της κλήσης.

Τα modules στο state

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
module.docs.local_file.css
module.docs.local_file.index
module.site.local_file.css
module.site.local_file.index
```

Κάθε resource έχει τη διεύθυνση `module.ΚΛΗΣΗ.τύπος.όνομα`. Τα `local_file.index` και `local_file.css` υπάρχουν δύο φορές, μία ανά κλήση, χωρίς σύγκρουση. Το πρόθεμα τα ξεχωρίζει. Αν ένα child module

καλούσε κι αυτό module, η διεύθυνση θα είχε δύο προθέματα, για παράδειγμα `module.site.module.assets.local_file.logo`. Με την πλήρη διεύθυνση δουλεύει και το `state show` του κεφαλαίου 4.

TERMINAL

```
terraform state show module.docs.local_file.css
```

ΕΞΟΔΟΣ

```
# module.docs.local_file.css:
resource "local_file" "css" {
    content              = <<-EOT
        h1 { color: teal; }
    EOT
    …
    directory_permission = "0777"
    file_permission      = "0777"
    filename             = "public/docs/style.css"
    id                   = "0d6486781fa3970a1ca242653deae5e790508e31"
}
```

Στο JSON του `terraform show -json` τα child modules βρίσκονται στο κλαδί `child_modules` του `root_module`, όπως είχε πει το κεφάλαιο 4.

TERMINAL

```
terraform show -json | jq -c '.values.root_module.child_modules[] | {address, n: (.resources | length)}'
```

ΕΞΟΔΟΣ

```
{"address":"module.docs","n":2}
{"address":"module.site","n":2}
```

Τα outputs του child δεν μπαίνουν στη λίστα του `terraform output`.

TERMINAL

```
terraform output
```

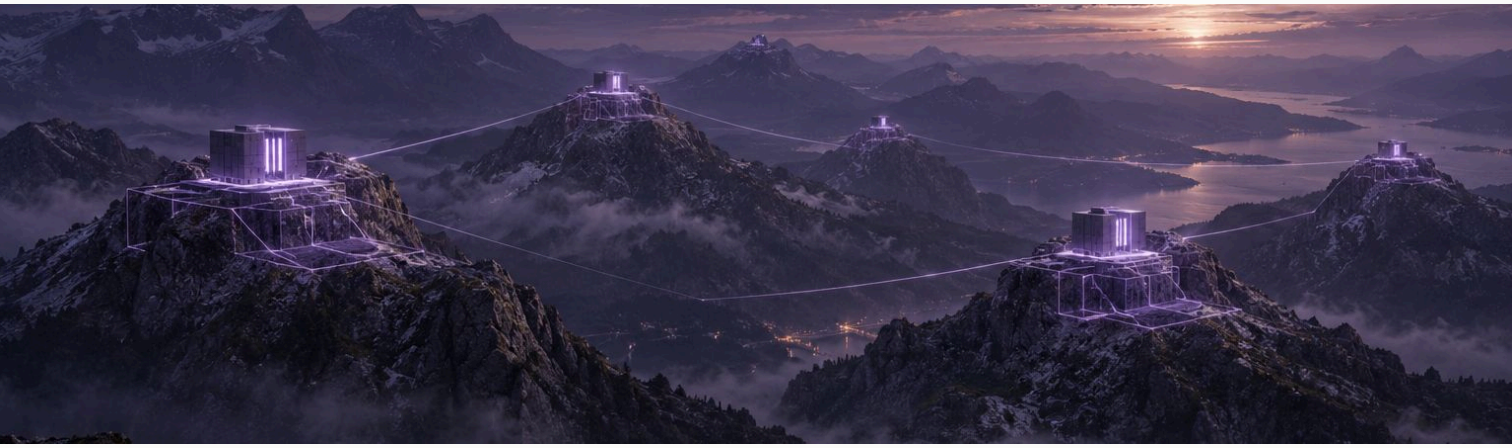
ΕΞΟΔΟΣ

```
blog_index = "public/blog/index.html"
docs_index = "public/docs/index.html"
```

Φαίνονται μόνο τα `blog_index` και `docs_index`, τα outputs του root module. Τα `index_path` και `files` του child φτάνουν μέχρι το root και σταματούν εκεί. Για να τα δεις στην οθόνη ή σε script, τα βγάζεις με δικό σου output, όπως έκανες με το `blog_index`.

ΚΡΑΤΑ ΑΥΤΟ

Η διεύθυνση ενός resource σε module είναι `module.ΚΛΗΣΗ.τύπος.όνομα`. Το label της κλήσης είναι μέρος της. Δύο κλήσεις του ίδιου module έχουν διαφορετικά labels, άρα και διαφορετικές διευθύνσεις.



Πολλές κλήσεις με for_each και count

Τρεις ιστοσελίδες θέλουν τρία blocks `module`. Όπως τα resources στο κεφάλαιο 11, ένα block `module` δέχεται `for_each` και `count`. Φτιάξε νέο φάκελο δίπλα στο `web`.

TERMINAL

```
cd ..
mkdir sites
cd sites
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

variable "sites" {
  type = map(string)
  default = {
    blog = "Το blog μου"
    docs = "Τεκμηρίωση"
    shop = "Το κατάστημα"
  }
}

module "site" {
  source   = "../web/modules/site"
  for_each = var.sites

  title      = each.value
  output_dir = "public/${each.key}"
}

output "indexes" {
```

```
  value = { for name, site in module.site : name => site.index_path }
}
```

Το `source` δείχνει στο module του φακέλου `web`. Τα `../` μετράνε από τον φάκελο `sites`, εκεί όπου βρίσκεται η κλήση. Το ίδιο module υπηρετεί πλέον δύο configurations. Το `for_each` δουλεύει όπως στο κεφάλαιο 11. Για κάθε κλειδί του map φτιάχνει ένα instance του module. Μέσα στα arguments τα `each.key` και `each.value` δίνουν τις τιμές του instance.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
Initializing modules...
- site in ../web/modules/site
…
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Plan: 6 to add, 0 to change, 0 to destroy.
…
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
…
```

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
module.site["blog"].local_file.css
module.site["blog"].local_file.index
module.site["docs"].local_file.css
module.site["docs"].local_file.index
module.site["shop"].local_file.css
module.site["shop"].local_file.index
```

Το κλειδί μπαίνει μετά το όνομα της κλήσης, για παράδειγμα `module.site["blog"].local_file.css`. Το `module.site` είναι πλέον map από instances, όπως ένα resource με `for_each`. Ένα instance το διαβάζεις με το κλειδί του.

TERRAFORM CONSOLE

```
> module.site["docs"].files
[
  "public/docs/index.html",
  "public/docs/style.css",
]
```

Το output `indexes` διατρέχει όλο το map με έκφραση `for` του κεφαλαίου 10.

TERMINAL

```
terraform output indexes
```

ΕΞΟΔΟΣ

```
{
  "blog" = "public/blog/index.html"
  "docs" = "public/docs/index.html"
  "shop" = "public/shop/index.html"
}
```

ΠΑΓΙΔΑ

Σε module με `for_each` το `module.site.index_path` δεν υπάρχει. Το Terraform σταματά στο plan με `Unsupported attribute` και γράφει `module.site is object with 3 attributes`. Τα τρία attributes είναι τα κλειδιά `blog`, `docs` και `shop`. Γράψε `module.site["blog"].index_path` ή διάτρεξε το map με `for`.

Το `count` ταιριάζει για το «ένα ή κανένα» του κεφαλαίου 11. Ένα module που υπάρχει μόνο όταν το ζητήσεις γράφεται έτσι.

HCL

```
module "preview" {
  source = "../web/modules/site"
  count  = var.with_preview ? 1 : 0

  title      = "Προεπισκόπηση"
  output_dir = "public/preview"
}
```

Με `with_preview = true` οι διευθύνσεις γίνονται `module.preview[0].local_file.index` και το output διαβάζεται ως `module.preview[0].index_path`. Με `false` το module δεν έχει κανένα instance και κανένα resource.

Πρόσεξε και το `modules.json` του φακέλου `sites`.

TERMINAL

```
jq -c '.Modules[]' .terraform/modules/modules.json
```

ΕΞΟΔΟΣ

```
{"Key":"","Source":"","Dir":"."}
{"Key":"site","Source":"../web/modules/site","Dir":"../web/modules/site"}
```

Η κλήση `site` είναι μία εγγραφή, όσα instances κι αν έχει. Ο κώδικας είναι ένας και τον διαβάζει από το `../web/modules/site`. Αλλάζουν μόνο οι τιμές κάθε instance.

Providers και modules

Ένα child module δηλώνει στο `required_providers` ποιους providers χρειάζεται και ποια ελάχιστη έκδοση, όπως το `versions.tf` του `site`. Δεν γράφει block `provider`. Τη ρύθμιση του provider την παίρνει από το root module. Ένα `provider "aws"` με `region = "eu-central-1"` στο root ισχύει για τα resources του AWS σε όλα τα modules του δέντρου, όπως θα δεις στο κεφάλαιο 17. Ο `local` δεν έχει ρυθμίσεις, αλλά η ιεραρχία φαίνεται στο `terraform providers` του κεφαλαίου 5.

TERMINAL

```
terraform providers
```

ΕΞΟΔΟΣ

```
Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/local] 2.9.1
└── module.site
    └── provider[registry.terraform.io/hashicorp/local] >= 2.9.1

Providers required by state:

    provider[registry.terraform.io/hashicorp/local]
```

Η τελεία είναι το root module με το `2.9.1`. Από κάτω κρέμεται το `module.site` με το δικό του `>= 2.9.1`. Η εντολή απαντά στο ερώτημα «ποιο module ζητά ποια έκδοση», που γίνεται σημαντικό όταν τα modules είναι πολλά.

Ένα module με δικό του block `provider` λέγεται legacy module. Το Terraform το δέχεται για λόγους συμβατότητας, με δύο συνέπειες. Η κλήση του δεν επιτρέπει `count`, `for_each` και `depends_on` και σταματά με `Module is incompatible with count, for_each, and depends_on`. Επίσης, αν σβήσεις την κλήση από τη configuration, φεύγει μαζί της και η ρύθμιση του provider. Το Terraform δεν έχει τότε provider για να σβήσει τα resources του module και το plan σταματά με `Provider configuration not present`. Γι' αυτό ένα module που θα ξαναχρησιμοποιηθεί δεν έχει ποτέ block `provider`.

Όταν το root έχει πολλές ρυθμίσεις του ίδιου provider με `alias`, όπως στο κεφάλαιο 5, η κλήση διαλέγει ποια θα πάρει το child με το meta-argument `providers`. Φτιάξε δίπλα στο `sites` ένα module που χρησιμοποιεί δύο providers, τον `random` και τον `local`.

TERMINAL

```
cd ..
mkdir -p pets/modules/pet
cd pets
```

modules/pet/main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = ">= 3.9.1"
    }
    local = {
      source  = "hashicorp/local"
      version = ">= 2.9.1"
    }
  }
}

resource "random_pet" "this" {}

resource "local_file" "name" {
  filename = "pet.txt"
  content  = "${random_pet.this.id}\n"
}
```

Το root έχει δεύτερη ρύθμιση του `random` με alias `second` και τη δίνει στο module.

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

provider "random" {
  alias = "second"
}

module "pet" {
  source = "./modules/pet"

  providers = {
    random = random.second
    local  = local
  }
}
```

Το `providers` είναι map. Το κλειδί είναι το τοπικό όνομα του provider μέσα στο child και η τιμή μια ρύθμιση του parent, χωρίς εισαγωγικά. Το `random.second` είναι η ρύθμιση με alias. Το σκέτο `local` είναι η προεπιλεγμένη ρύθμιση του `local` στο root. Για το child και οι δύο είναι απλώς ο `random` και ο `local` του.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

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

TERMINAL

```
grep '"provider"' terraform.tfstate
```

ΕΞΟΔΟΣ

```
      "provider": "provider[\"registry.terraform.io/hashicorp/local\"]",
      "provider": "provider[\"registry.terraform.io/hashicorp/random\"].second",
```

Το state σημειώνει ποια ρύθμιση έφτιαξε κάθε resource, όπως στο κεφάλαιο 5. Το `local_file` φτιάχτηκε με την προεπιλεγμένη ρύθμιση του `local`. Το `random_pet` έχει το `.second` στο τέλος, άρα φτιάχτηκε με τη ρύθμιση `random.second`, αν και μέσα στο module δεν γράφει πουθενά `provider`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Οι κανόνες για τους providers ενός child module είναι τρεις.

- Χωρίς `providers` στην κλήση, το child κληρονομεί αυτόματα την προεπιλεγμένη ρύθμιση κάθε provider του parent, το block χωρίς `alias`.
- Μια ρύθμιση με `alias` δεν κληρονομείται ποτέ αυτόματα. Περνά μόνο ρητά, με το `providers`.
- Μόλις η κλήση γράψει `providers`, η τεκμηρίωση του Terraform λέει ότι η αυτόματη κληρονομιά ακυρώνεται και το child έχει μόνο τις ρυθμίσεις του map. Γι' αυτό το map απαριθμεί όλους τους providers του child. Η κλήση `pet` γράφει και το `local = local`, αν και για τον `local` δεν αλλάζει τίποτα.

ΣΗΜΕΙΩΣΗ

Τα Terraform 1.12 έως 1.16 δεν επιβάλλουν τον τρίτο κανόνα. Αν σβήσεις τη γραμμή `local = local`, δεν βγαίνει ούτε προειδοποίηση και το `local_file` παίρνει πάλι την προεπιλεγμένη ρύθμιση του root. Το ίδιο συμβαίνει με provider που έχει ρυθμίσεις, όπως το `region` του `aws`. Μη βασίζεσαι σε αυτό. Όταν γράφεις `providers`, βάζε στο map κάθε provider που χρησιμοποιεί το child. Στην εξέταση ισχύει ο κανόνας της τεκμηρίωσης.

Η συνηθισμένη χρήση του `providers` είναι το ίδιο module σε δύο regions του AWS, στο κεφάλαιο 17.

depends_on σε module

Το `depends_on` του κεφαλαίου 6 μπαίνει και σε block `module`.

```hcl
module "site" {
  source = "./modules/site"

  title      = "Το blog μου"
  output_dir = "public/blog"

  depends_on = [local_file.robots]
}
```

Κάθε resource του module περιμένει τώρα το `local_file.robots`. Η εξάρτηση είναι συνολική. Αφορά όλο το module και όχι μόνο όσα resources χρειάζονται πραγματικά το `robots`. Όπως και στα resources, προτίμησε μια αναφορά. Αν δώσεις σε μια μεταβλητή του module μια τιμή από το `local_file.robots`, η εξάρτηση δημιουργείται μόνη της και αφορά μόνο όσα resources χρησιμοποιούν αυτή τη μεταβλητή.

Το block `module` έχει λοιπόν δύο είδη arguments.

Argument	Ρόλος
`source`	πού βρίσκεται ο κώδικας, υποχρεωτικό
`version`	έκδοση για modules από registry, κεφάλαιο 16
`count`, `for_each`	πολλά instances από ένα block
`providers`	ποιες ρυθμίσεις providers παίρνει το child
`depends_on`	ρητή εξάρτηση για όλο το module
κάθε άλλο όνομα	τιμή για τη μεταβλητή του child με το ίδιο όνομα

Καλές πρακτικές

Ένα module αξίζει όταν κάνει τον κώδικα πιο απλό για όποιον το καλεί. Οι παρακάτω κανόνες βοηθούν.

- **Ένας σκοπός.** Ένα module για την ιστοσελίδα, ένα για το δίκτυο. Αν δεν περιγράφεις τι κάνει σε μία πρόταση, χώρισέ το.
- **Όχι περιτύλιγμα ενός resource.** Ένα module που τυλίγει ένα `local_file` και περνά όλα τα arguments του ένα προς ένα προσθέτει επίπεδο χωρίς αξία.
- **Η συνηθισμένη δομή.** `main.tf`, `variables.tf`, `outputs.tf`, `versions.tf` και ένα `README.md` που δείχνει μια κλήση. Το κεφάλαιο 16 δείχνει τη δομή ενός module που δημοσιεύεται.
- **Τεκμηρίωση στον κώδικα.** `type` και `description` σε κάθε μεταβλητή, `description` σε κάθε output. Είναι ό,τι διαβάζει πρώτα όποιος καλεί το module.
- **Λογικά defaults.** Υποχρεωτικό ό,τι αλλάζει από κλήση σε κλήση, `default` για ό,τι ταιριάζει στις περισσότερες.
- **Αρκετά outputs.** Ό,τι δεν βγάζει το module δεν φτάνει ποτέ στο root. Δώσε όσα θα χρειαστεί ο καλών, όπως paths, ονόματα και IDs.
- **Providers από το root.** `required_providers` με `>=` και κανένα block `provider`.
- **Ρηχό δέντρο.** Ένα ή δύο επίπεδα modules διαβάζονται εύκολα. Πέντε επίπεδα κάνουν κάθε αλλαγή κυνήγι.

Ασκήσεις

Οι ασκήσεις χρησιμοποιούν το module `site` του φακέλου `web`. Όσες λένε ότι ξεκινούν από το `web` ξεκινούν από αντίγραφό του, μαζί με το `modules/site`, χωρίς state και χωρίς `.terraform`. Οι υπόλοιπες ξεκινούν σε κενό φάκελο μέσα στο `~/tfbook/ch15` και καλούν το module με `source = "../web/modules/site"`.

ΑΣΚΗΣΗ 15.1 Τρίτη ιστοσελίδα

Ξεκίνα από το `web`. Πρόσθεσε στο `main.tf` κλήση με όνομα `shop`, τίτλο `Το κατάστημα`, φάκελο `public/shop` και χρώμα `green`. Τρέξε init, apply και `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
module.docs.local_file.css
module.docs.local_file.index
module.shop.local_file.css
module.shop.local_file.index
module.site.local_file.css
module.site.local_file.index
```

ΑΣΚΗΣΗ 15.2 Ένα output ακόμη

Ξεκίνα από το `web`. Πρόσθεσε στο module output `css_path` με το path του stylesheet και `description`. Πρόσθεσε στο root output `docs_css` που δίνει το `css_path` της κλήσης `docs`. Κάνε apply και τύπωσε την τιμή με `terraform output -raw docs_css`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
public/docs/style.css
```

ΑΣΚΗΣΗ 15.3 Η πόρτα του module

Ξεκίνα από το `web`. Γράψε τα δύο αρχεία που ακολουθούν. Το root έχει τη μεταβλητή `owner` και το module θέλει να τη χρησιμοποιήσει στο `footer.html`.

owner.tf

```
variable "owner" {
  type    = string
  default = "Μαρία"
}
```

modules/site/footer.tf

```
resource "local_file" "footer" {
  filename = "${var.output_dir}/footer.html"
  content  = "<footer>${var.owner}</footer>\n"
}
```

Τρέξε `terraform validate` και διάβασε το σφάλμα. Διόρθωσε τη configuration χωρίς να σβήσεις τη μεταβλητή του root, ώστε και οι δύο ιστοσελίδες να πάρουν footer με την τιμή της. Κάνε apply και δείξε το `public/blog/footer.html`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
<footer>Μαρία</footer>
```

ΑΣΚΗΣΗ 15.4 Ιστοσελίδες από map

Σε νέο φάκελο κάλεσε το module με όνομα `site` και `for_each` πάνω σε map με κλειδιά `el` και `en` και τιμές `Καλώς ήρθες` και `Welcome`. Ο τίτλος κάθε ιστοσελίδας είναι η τιμή και ο φάκελος είναι το `public/` ακολουθούμενο από το κλειδί. Κάνε apply και δείξε τις διευθύνσεις με `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
module.site["el"].local_file.css
module.site["el"].local_file.index
module.site["en"].local_file.css
module.site["en"].local_file.index
```

ΑΣΚΗΣΗ 15.5 Ιστοσελίδα κατά παραγγελία

Σε νέο φάκελο γράψε μεταβλητή `with_preview` τύπου `bool` με default `false` και κλήση `preview` του module με `count`, τίτλο `Προεπισκόπηση` και φάκελο `public/preview`. Κάνε apply μία φορά χωρίς τιμές και μία φορά με `-var with_preview=true`. Δείξε τη λίστα του state μετά το δεύτερο apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
module.preview[0].local_file.css
module.preview[0].local_file.index
```

ΑΣΚΗΣΗ 15.6 Ποιος ζητά ποια έκδοση

Σε νέο φάκελο γράψε ένα module στο `modules/name` που ζητά τον `hashicorp/random` με constraint `>= 3.9.1`, έχει ένα `random_pet` και output `name`. Το root δηλώνει τον `random` με έκδοση `3.9.1` και καλεί το module με όνομα `pet`. Τρέξε init και `terraform providers`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/random] 3.9.1
└── module.pet
    └── provider[registry.terraform.io/hashicorp/random] >= 3.9.1
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 15.1

Ποια πρόταση περιγράφει σωστά το root module;

a. Το module που βρίσκεται στον φάκελο `modules` ενός project.
b. Τα αρχεία `.tf` του φακέλου όπου τρέχεις τις εντολές του Terraform.
c. Ένα module δημοσιευμένο στο Terraform Registry.
d. Το πρώτο block `module` του αρχείου `main.tf`.

ΕΡΩΤΗΣΗ 15.2

Ένα child module που καλείται με `module "network"` έχει output με όνομα `vpc_id`. Πώς το διαβάζεις στο root module;

a. `var.network.vpc_id`
b. `output.network.vpc_id`
c. `module.network.vpc_id`
d. `module.network.outputs.vpc_id`

ΕΡΩΤΗΣΗ 15.3

Το root module έχει τη μεταβλητή `environment`. Ένα child module χρειάζεται την τιμή της. Τι κάνεις;

a. Δηλώνεις `variable "environment"` στο child και γράφεις `environment = var.environment` στο block `module`.
b. Τίποτα. Το child διαβάζει αυτόματα το `var.environment` του root.
c. Γράφεις `var.root.environment` μέσα στο child.
d. Μετακινείς τη μεταβλητή σε `locals` του root, που τα βλέπουν όλα τα modules.

ΕΡΩΤΗΣΗ 15.4

Πρόσθεσες νέο block `module` με τοπικό `source` και το `terraform plan` σταματά με `Module not installed`. Ποιες εντολές λύνουν το πρόβλημα; Επίλεξε δύο.

a. `terraform init`
b. `terraform validate`
c. `terraform get`
d. `terraform providers`
e. `terraform apply -auto-approve`

ΕΡΩΤΗΣΗ 15.5

Σωστό ή λάθος; Ένα module που θα το χρησιμοποιούν πολλές configurations πρέπει να περιέχει δικό του block `provider` με όλες τις ρυθμίσεις του provider.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 15.6

Το block `module "app"` έχει `for_each = toset(["a", "b"])` και το module περιέχει το `resource "local_file" "conf"`. Ποια είναι η διεύθυνση του resource για το κλειδί `b`;

a. `local_file.conf["b"]`
b. `module.app.local_file.conf["b"]`
c. `module["b"].app.local_file.conf`
d. `module.app["b"].local_file.conf`



ΚΕΦΑΛΑΙΟ 16

Πηγές modules, εκδόσεις και Registry

Στο κεφάλαιο 15 κάθε module ήταν ένας φάκελος δίπλα στη configuration σου. Τα περισσότερα modules όμως τα γράφουν άλλοι και ζουν αλλού, σε ένα registry ή σε ένα Git repository. Σε αυτό το κεφάλαιο τα φέρνεις από εκεί, καρφώνεις την έκδοσή τους και βλέπεις τι ακριβώς κατέβηκε στον δίσκο σου.

Από πού έρχεται ένα module

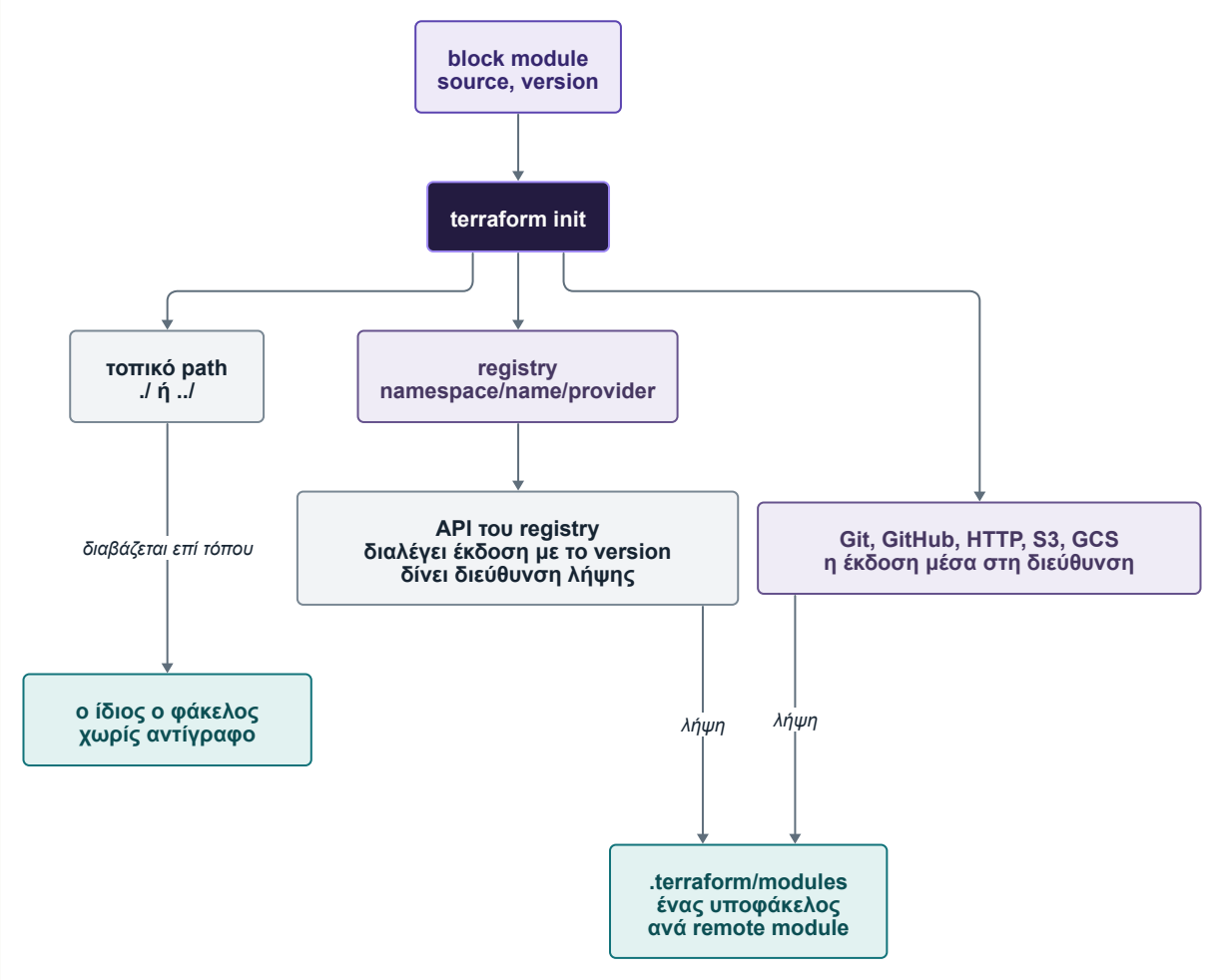
Κάθε block `module` έχει ένα υποχρεωτικό argument, το `source`. Λέει από πού θα έρθει ο κώδικας του module. Το `terraform init` διαβάζει κάθε `source`, φέρνει τον κώδικα και σημειώνει τι έφερε στον φάκελο `.terraform/modules`. Το plan και το apply δεν κατεβάζουν ποτέ modules. Δουλεύουν μόνο με όσα βρήκε εκεί το init.

Από τη μορφή του `source` το Terraform καταλαβαίνει ποιον τρόπο εγκατάστασης θα χρησιμοποιήσει. Ο πίνακας δείχνει τις πηγές που θα συναντήσεις.

Πηγή	Παράδειγμα `source`	Η έκδοση ορίζεται με
Τοπικό path	`./modules/site`	δεν έχει έκδοση
Terraform Registry	`hashicorp/dir/template`	`version`
Private registry	`app.terraform.io/acme/vpc/aws`	`version`
Git	`git::https://example.com/vpc.git?ref=v1.2.0`	`?ref=`
GitHub	`github.com/acme/terraform-aws-vpc?ref=v1.2.0`	`?ref=`
HTTP	`https://example.com/vpc-1.2.0.zip`	το ίδιο το URL

Πηγή	Παράδειγμα `source`	Η έκδοση ορίζεται με
S3 και GCS	`s3::https://...`, `gcs::https://...`	το ίδιο το URL

Τα τοπικά paths διαβάζονται επί τόπου. Όλες οι άλλες πηγές λέγονται **remote**. Το init τις κατεβάζει σε έναν υποφάκελο του `.terraform/modules` για κάθε block `module`. Το argument `version` υπάρχει μόνο για τα registries. Στις άλλες πηγές η έκδοση γράφεται μέσα στο ίδιο το `source`.



Σχήμα 16.1 · Πώς το init χειρίζεται κάθε πηγή. Ένα τοπικό path διαβάζεται εκεί που βρίσκεται. Για ένα registry το init ρωτά πρώτα το API ποια έκδοση ταιριάζει και από πού κατεβαίνει. Οι υπόλοιπες πηγές κατεβαίνουν απευθείας από τη διεύθυνση που γράφεις.

Τοπικό path

Το κεφάλαιο 15 χρησιμοποίησε μόνο τοπικά paths. Ένα τοπικό path ξεκινά πάντα με `./` ή `../`. Το πρόθεμα δεν είναι διακοσμητικό. Από αυτό το Terraform καταλαβαίνει ότι μιλάς για φάκελο. Φτιάξε ένα μικρό module και κάλεσέ το χωρίς πρόθεμα για να δεις τι γίνεται.

TERMINAL

```
mkdir -p paths/modules/hello
cd paths
```

modules/hello/main.tf

```
variable "name" {
  type = string
}

output "message" {
  value = "Γεια σου, ${var.name}!"
}
```

main.tf

```
module "hello" {
  source = "modules/hello"

  name = "Terraform"
}

output "message" {
  value = module.hello.message
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
╷
│ Error: Invalid module source address
│
│   on main.tf line 2, in module "hello":
│    2:   source = "modules/hello"
│
│ Terraform failed to determine your intended installation method for remote
│ module package "modules/hello".
│
│ If you intended this as a path relative to the current module, use
│ "./modules/hello" instead. The "./" prefix indicates that the address is a
│ relative filesystem path.
╵
```

Το σφάλμα είναι `Invalid module source address`. Χωρίς `./` το Terraform θεώρησε το `modules/hello` διεύθυνση remote πακέτου. Δοκίμασε όλες τις μορφές remote πηγών που ξέρει και καμία δεν ταίριαξε. Το ίδιο το μήνυμα σου λέει τη διόρθωση, το `"./modules/hello"`.

main.tf

```
module "hello" {
  source = "./modules/hello"

  name = "Terraform"
}

output "message" {
  value = module.hello.message
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
- hello in modules/hello

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Η ενότητα `Initializing modules...` γράφει τώρα μόνο `- hello in modules/hello`. Δεν υπάρχει γραμμή `Downloading`. Το Terraform σημειώνει όσα modules εγκατέστησε στο αρχείο `.terraform/modules/modules.json`. Το `jq`, που είδες στο κεφάλαιο 4, το τυπώνει ευανάγνωστο.

TERMINAL

```
jq . .terraform/modules/modules.json
```

ΕΞΟΔΟΣ

```
{
  "Modules": [
    {
      "Key": "",
      "Source": "",
      "Dir": "."
    },
    {
      "Key": "hello",
      "Source": "./modules/hello",
      "Dir": "modules/hello"
    }
  ]
}
```

Το module `hello` έχει `Dir` τον φάκελο `modules/hello`. Το Terraform δεν έφτιαξε αντίγραφο. Διαβάζει τα αρχεία εκεί που βρίσκονται, οπότε μια αλλαγή μέσα στο `modules/hello` φαίνεται στο επόμενο plan χωρίς νέο init. Η πρώτη εγγραφή, με κενό `Key`, είναι το root module.

Ένα τοπικό module δεν έχει εκδόσεις. Είναι ό,τι υπάρχει στον φάκελο αυτή τη στιγμή. Δες τι γίνεται αν του δώσεις `version`.

main.tf

```
module "hello" {
  source  = "./modules/hello"
  version = "1.0.0"

  name = "Terraform"
}

output "message" {
  value = module.hello.message
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
╷
│ Error: Invalid registry module source address
│
│   on main.tf line 2, in module "hello":
│    2:   source  = "./modules/hello"
│
│ Failed to parse module registry address: can't use local directory
│ "./modules/hello" as a module registry address.
│
│ Terraform assumed that you intended a module registry source address
│ because you also set the argument "version", which applies only to registry
│ modules.
╵
```

Το μήνυμα εξηγεί τη λογική του Terraform. Είδε `version` και υπέθεσε ότι το `source` είναι διεύθυνση registry, γιατί το `version` ισχύει μόνο για modules από registry. Μετά απέτυχε να διαβάσει το `./modules/hello` ως τέτοια διεύθυνση. Για ένα τοπικό module η έκδοση είναι η έκδοση του repository που το περιέχει. Αν θέλεις να κρατάς εκδόσεις ενός module, το βάζεις σε δικό του repository ή σε registry.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Ένα τοπικό path ξεκινά με `./` ή `../`. Δεν κατεβαίνει, δεν αντιγράφεται και δεν δέχεται `version`. Η εξέταση ρωτά συχνά ποιες πηγές δέχονται `version`. Η απάντηση είναι μόνο τα registries, το δημόσιο Terraform Registry και ένα private registry.

Ένα module από το Terraform Registry

Το Terraform Registry του κεφαλαίου 5 δεν έχει μόνο providers. Έχει και χιλιάδες modules. Η διεύθυνση ενός module στο Registry έχει τρία μέρη και ένα προαιρετικό τέταρτο μπροστά.

ΕΞΟΔΟΣ

```
[HOSTNAME/]NAMESPACE/NAME/PROVIDER
```

- Το **NAMESPACE** είναι ο οργανισμός ή ο χρήστης που δημοσιεύει το module.
- Το **NAME** είναι το όνομα του module.
- Το **PROVIDER** είναι η πλατφόρμα που αφορά κυρίως το module, για παράδειγμα `aws` ή `azurerm`.
- Το **HOSTNAME** λείπει για το δημόσιο Registry, το `registry.terraform.io`. Ένα private registry το γράφει.

Πρόσεξε ότι η διεύθυνση ενός provider έχει δύο μέρη, `hashicorp/local`, ενώ ενός module τρία. Από τον αριθμό των μερών το Terraform ξεχωρίζει ένα module του Registry από τις άλλες πηγές.

Θα χρησιμοποιήσεις το `hashicorp/dir/template`. Το δημοσιεύει η HashiCorp και κάνει κάτι απλό. Διαβάζει όλα τα αρχεία ενός φακέλου. Όσα τελειώνουν σε `.tmpl` τα περνά από την `templatefile` του κεφαλαίου 10 και επιστρέφει το αποτέλεσμα. Για τα υπόλοιπα επιστρέφει μόνο το path τους. Δεν έχει resources και δεν χρειάζεται provider, οπότε δουλεύει χωρίς cloud. Το τρίτο μέρος, `template`, ήρθε από τον παλιό provider `template`, που είχε ένα resource για την ίδια δουλειά. Το module το αντικατέστησε.

Φτιάξε έναν φάκελο με μια μικρή ιστοσελίδα. Ένα template και ένα στατικό αρχείο.

TERMINAL

```
cd ..
mkdir -p site/src
cd site
```

src/index.html.tmpl

```
<h1>${title}</h1>
<p>Χτίστηκε με Terraform.</p>
```

src/style.css

```
body { font-family: sans-serif; }
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

module "site" {
  source  = "hashicorp/dir/template"
  version = "1.0.2"

  base_dir = "${path.module}/src"
  template_vars = {
    title = "Η σελίδα μου"
  }
}

resource "local_file" "public" {
  for_each = module.site.files

  filename = "public/${each.key}"
  content  = each.value.content
  source   = each.value.source_path
}

output "types" {
  value = { for path, file in module.site.files : path => file.content_type }
}
```

Διάβασε το block `module` γραμμή γραμμή.

- Το `source` έχει τρία μέρη χωρίς πρόθεμα, άρα είναι διεύθυνση του δημόσιου Registry.
- Το `version` καρφώνει την έκδοση 1.0.2, όπως καρφώνεις κάθε provider.
- Το `base_dir` είναι ο φάκελος που θα διαβάσει το module. Το `path.module` του κεφαλαίου 10 τον κάνει σχετικό με τον φάκελο της configuration.
- Το `template_vars` δίνει τις τιμές που χρησιμοποιούν τα templates, εδώ το `title`.

Το output `files` του module είναι ένα map με κλειδί το σχετικό path κάθε αρχείου. Κάθε τιμή έχει `content` για όσα αρχεία ήταν templates και `source_path` για τα στατικά. Το άλλο από τα δύο είναι `null`. Το resource `local_file` δέχεται είτε `content` είτε `source`, που αντιγράφει ένα υπάρχον αρχείο. Με το `for_each` του κεφαλαίου 11 φτιάχνεις ένα αρχείο στο `public` για κάθε αρχείο του `src`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
Downloading registry.terraform.io/hashicorp/dir/template 1.0.2 for site...
- site in .terraform/modules/site
```

```
Initializing provider plugins...
- Finding hashicorp/local versions matching "2.9.1"...
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Η ενότητα `Initializing modules...` τρέχει πριν από τους providers. Η γραμμή `Downloading` γράφει την πλήρη διεύθυνση με το hostname, `registry.terraform.io/hashicorp/dir/template`, την έκδοση 1.0.2 και το όνομα του block, `site`. Η επόμενη γραμμή λέει πού μπήκε ο κώδικας.

TERMINAL

```
ls -a .terraform/modules/site
```

ΕΞΟΔΟΣ

```
.
..
.git
CHANGELOG.md
files.tf
LICENSE
outputs.tf
README.md
test
variables.tf
versions.tf
```

Εδώ ζει τώρα το module. Τα inputs και τα outputs του τα βρίσκεις στα αρχεία του.

TERMINAL

```
grep -h -E "^(variable|output)" .terraform/modules/site/*.tf
```

ΕΞΟΔΟΣ

```
output "files" {
output "files_on_disk" {
output "files_in_memory" {
variable "base_dir" {
variable "template_vars" {
variable "template_file_suffix" {
variable "file_types" {
variable "default_file_type" {
```

Είναι πέντε inputs και τρία outputs. Στο `variables.tf` μόνο το `base_dir` δεν έχει `default`, άρα είναι το μόνο υποχρεωτικό input. Το `template_file_suffix` αλλάζει την κατάληξη των templates από `.tmpl` σε ό,τι θέλεις. Τα `file_types` και `default_file_type` λένε τι `content_type` παίρνει κάθε κατάληξη. Εκτός από το `files` υπάρχουν τα `files_on_disk` και `files_in_memory`, που κρατούν μόνο τα στατικά ή μόνο τα templates.

TERMINAL

```
cat .terraform/modules/site/versions.tf
```

ΕΞΟΔΟΣ

```
terraform {
  required_version = ">= 0.12.8"
}
```

Το `versions.tf` του module έχει μόνο `required_version` και κανένα `required_providers`. Το module δεν χρειάζεται provider, γιατί δουλεύει μόνο με συναρτήσεις. Ο `local` που κατέβηκε τον ζήτησε η δική σου configuration.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.public["index.html"] will be created
  + resource "local_file" "public" {
      + content              = <<-EOT
            <h1>Η σελίδα μου</h1>
            <p>Χτίστηκε με Terraform.</p>
        EOT
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "public/index.html"
      + id                   = (known after apply)
    }

  # local_file.public["style.css"] will be created
  + resource "local_file" "public" {
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
```

```
      + filename             = "public/style.css"
      + id                   = (known after apply)
      + source               = "./src/style.css"
    }

Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + types = {
      + "index.html" = "text/html; charset=utf-8"
      + "style.css"  = "text/css; charset=utf-8"
    }
local_file.public["index.html"]: Creating...
local_file.public["style.css"]: Creating...
local_file.public["style.css"]: Creation complete after 0s [id=ca74c069ae39ff7cb59538bf7271782dde2a172b]
local_file.public["index.html"]: Creation complete after 0s [id=b3b3f031086da620068621cc95efefe3f03ca985]

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

Outputs:

types = {
  "index.html" = "text/html; charset=utf-8"
  "style.css" = "text/css; charset=utf-8"
}
```

Το plan έχει δύο resources, `local_file.public["index.html"]` και `local_file.public["style.css"]`. Το πρώτο έχει `content` με το template γεμάτο. Το δεύτερο έχει `source` με το path του στατικού αρχείου. Το output `types` δείχνει ότι το module έδωσε σε κάθε αρχείο σωστό `content_type`.

TERMINAL

```
cat public/index.html
```

ΕΞΟΔΟΣ

```
<h1>Η σελίδα μου</h1>
<p>Χτίστηκε με Terraform.</p>
```

Το `${title}` έγινε `Η σελίδα μου`. Τον κώδικα που το έκανε δεν τον έγραψες εσύ. Τον κατέβασε το init από το Registry.

Πίσω από το Registry

Στο `ls -a` είδες ότι ο φάκελος του module έχει `.git`. Το Registry δεν φυλάει τον κώδικα των δημόσιων modules. Κρατά έναν κατάλογο με τις εκδόσεις και, για κάθε έκδοση, τη διεύθυνση από όπου κατεβαίνει. Το init ρωτά το Registry, παίρνει τη διεύθυνση και κάνει clone από εκεί. Ρώτα το Git από πού ήρθε ο κώδικας. Το `git -C φάκελος` τρέχει την εντολή σαν να ήσουν μέσα σε εκείνον τον φάκελο.

TERMINAL

```
git -C .terraform/modules/site remote get-url origin
git -C .terraform/modules/site describe --tags
```

ΕΞΟΔΟΣ

```
https://github.com/hashicorp/terraform-template-dir
v1.0.2
```

Ο κώδικας ήρθε από το repository `terraform-template-dir` στο GitHub. Το όνομα ακολουθεί τον κανόνα του δημόσιου Registry, `terraform-PROVIDER-NAME`. Από το `terraform-template-dir` το Registry βγάζει το `template` ως provider και το `dir` ως όνομα. Η έκδοση 1.0.2 είναι το Git tag `v1.0.2`. Κάθε νέο tag με semantic version γίνεται αυτόματα νέα έκδοση στο Registry.

Το argument version

Το `version` ενός module δέχεται τα ίδια version constraints με τους providers του κεφαλαίου 5, δηλαδή `=`, `!=`, `>`, `>=`, `<`, `<=` και `~>`. Το `hashicorp/dir/template` έχει τρεις εκδόσεις, την 1.0.0, την 1.0.1 και την 1.0.2. Ζήτα την 1.0.1 και τρέξε plan χωρίς init.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

module "site" {
  source  = "hashicorp/dir/template"
  version = "1.0.1"

  base_dir = "${path.module}/src"
  template_vars = {
    title = "Η σελίδα μου"
  }
}

resource "local_file" "public" {
  for_each = module.site.files

  filename = "public/${each.key}"
  content  = each.value.content
  source   = each.value.source_path
}

output "types" {
  value = { for path, file in module.site.files : path => file.content_type }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Module version requirements have changed
│
│   on main.tf line 11, in module "site":
│   11:   source  = "hashicorp/dir/template"
│
│ The version requirements have changed since this module was installed and
│ the installed version (1.0.2) is no longer acceptable. Run "terraform init"
│ to install all modules required by this configuration.
╵
```

Το σφάλμα είναι `Module version requirements have changed`. Η εγκατεστημένη έκδοση, η 1.0.2, δεν ταιριάζει πια με το `version`. Το Terraform δεν κατεβάζει μόνο του. Σου λέει να τρέξεις `terraform init`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
Downloading registry.terraform.io/hashicorp/dir/template 1.0.1 for site...
- site in .terraform/modules/site

…
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
local_file.public["index.html"]: Refreshing state... [id=b3b3f031086da620068621cc95efefe3f03ca985]
local_file.public["style.css"]: Refreshing state... [id=ca74c069ae39ff7cb59538bf7271782dde2a172b]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το init κατέβασε την 1.0.1 και το plan δεν βρίσκει αλλαγές. Η μόνη διαφορά της 1.0.2 αφορά τα αρχεία `.svg`, που δεν έχεις. Γράψε τώρα ένα χαλαρότερο constraint, το `~> 1.0`, που δέχεται κάθε 1.x.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
```

```
}

module "site" {
  source  = "hashicorp/dir/template"
  version = "~> 1.0"

  base_dir = "${path.module}/src"
  template_vars = {
    title = "Η σελίδα μου"
  }
}

resource "local_file" "public" {
  for_each = module.site.files

  filename = "public/${each.key}"
  content  = each.value.content
  source   = each.value.source_path
}

output "types" {
  value = { for path, file in module.site.files : path => file.content_type }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...

Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock file
…
```

Δεν υπάρχει γραμμή `Downloading`. Η εγκατεστημένη 1.0.1 ταιριάζει με το `~> 1.0`, οπότε το init την κρατά. Όπως στους providers, η σημαία `-upgrade` ζητά τη νεότερη έκδοση που επιτρέπει το constraint.

TERMINAL

```
terraform init -upgrade
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Upgrading modules...
Downloading registry.terraform.io/hashicorp/dir/template 1.0.2 for site...
- site in .terraform/modules/site

…
```

Η ενότητα λέγεται τώρα `Upgrading modules...` και κατέβηκε η 1.0.2. Το `terraform get` του κεφαλαίου 15 κάνει μόνο το κομμάτι των modules. Το `terraform get -update` είναι το αντίστοιχο του `init -upgrade`.

> **ΠΑΓΙΔΑ**
> Χωρίς `version` το init παίρνει τη νεότερη έκδοση του Registry τη στιγμή που τρέχει. Δεν σε προειδοποιεί. Δήλωνε πάντα `version` σε κάθε module του Registry.

Το lock file δεν κλειδώνει modules

Στο κεφάλαιο 5 το lock file κρατούσε την επιλογή του init για κάθε provider. Δες τι κράτησε τώρα.

TERMINAL

```
cat .terraform.lock.hcl
```

ΕΞΟΔΟΣ

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/local" {
  version     = "2.9.1"
  constraints = "2.9.1"
…
```

Υπάρχει μόνο το block `provider` του `local`. Για το module δεν γράφτηκε τίποτα. Η επιλογή 1.0.2 υπάρχει μόνο στο `modules.json`.

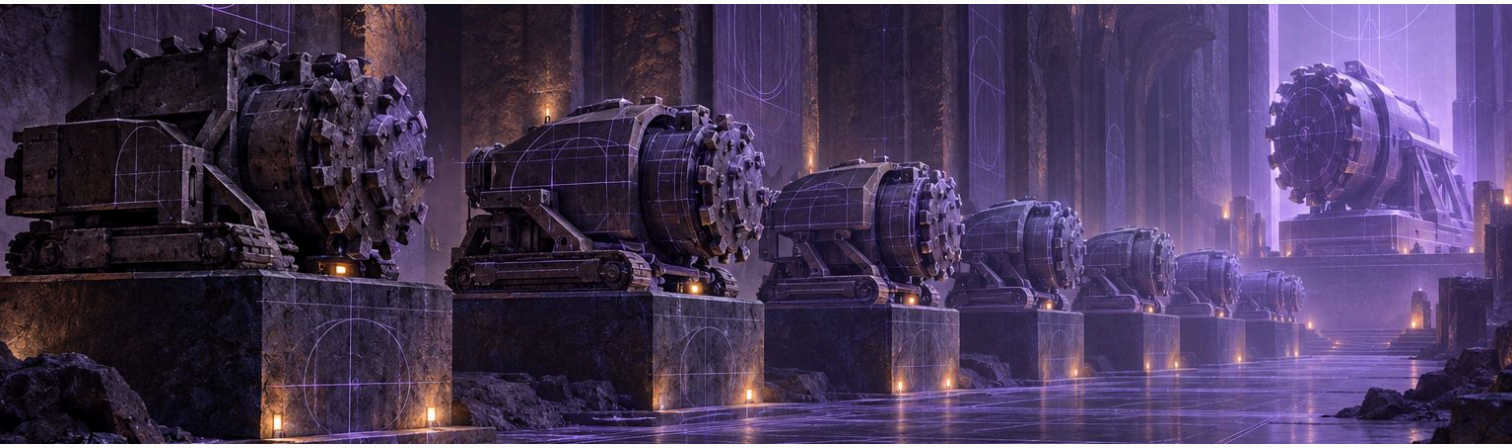
TERMINAL

```
jq . .terraform/modules/modules.json
```

ΕΞΟΔΟΣ

```
{
  "Modules": [
    {
      "Key": "",
      "Source": "",
      "Dir": "."
    },
    {
      "Key": "site",
      "Source": "registry.terraform.io/hashicorp/dir/template",
      "Version": "1.0.2",
      "Dir": ".terraform/modules/site"
    }
  ]
}
```

Το `modules.json` βρίσκεται μέσα στον `.terraform`, που δεν μπαίνει σε Git. Ένας συνάδελφος που κάνει clone το project ξεκινά χωρίς αυτό. Το init του διαλέγει τη νεότερη έκδοση που επιτρέπει το `version` εκείνη τη μέρα. Αν γράψεις `~> 1.0` και αύριο βγει μια 1.1.0, ο συνάδελφος θα πάρει την 1.1.0 ενώ εσύ έχεις ακόμη την 1.0.2.



ΚΡΑΤΑ ΑΥΤΟ

Το lock file κλειδώνει μόνο providers. Την έκδοση ενός module την κλειδώνεις εσύ στη configuration, με ακριβές `version` για το Registry ή με tag ή commit στο `ref` για το Git. Ένα χαλαρό constraint σε module σημαίνει ότι δύο μηχανήματα μπορεί να τρέχουν διαφορετικό κώδικα.

Modules από Git

Οι περισσότερες εταιρείες κρατούν τα modules τους σε Git repositories. Θα φτιάξεις ένα τοπικό repository με ένα module, ώστε να δουλεύεις χωρίς λογαριασμό σε κάποιον server. Το module σχηματίζει έναν χαιρετισμό.

TERMINAL

```
cd ~/tfbook/ch16
mkdir -p repos/greeting
cd repos/greeting
git init -b main
```

ΕΞΟΔΟΣ

```
Initialized empty Git repository in /home/you/tfbook/ch16/repos/greeting/.git/
```

Το `git init` έφτιαξε ένα κενό repository με κλαδί `main`. Γράψε το module χωρισμένο στα αρχεία που περιμένει κάθε χρήστης modules.

versions.tf

```
terraform {
  required_version = ">= 1.12.0"
}
```

variables.tf

```
variable "name" {
  type        = string
  description = "Ποιον χαιρετά το module."
}
```

main.tf

```
locals {
  message = "Γεια σου, ${var.name}!"
}
```

outputs.tf

```
output "message" {
  value       = local.message
  description = "Ο χαιρετισμός."
}
```

README.md

```
# greeting

Σχηματίζει έναν χαιρετισμό.

Inputs
- name, ποιον χαιρετά.

Outputs
- message, ο χαιρετισμός.
```

Κάνε commit και βάλε το tag `v1.0.0`. Οι σημαίες `-c` δίνουν όνομα και email μόνο για αυτό το commit. Αν έχεις ήδη ρυθμίσει το Git, μπορείς να τις παραλείψεις.

TERMINAL

```
git add .
git -c user.name=You -c user.email=you@example.com commit -q -m "Πρώτη έκδοση"
git tag v1.0.0
git tag
```

ΕΞΟΔΟΣ

```
v1.0.0
```

Τώρα κάλεσε το module από μια configuration σε άλλον φάκελο.

TERMINAL

```
cd ~/tfbook/ch16
mkdir hello
cd hello
```

Το `source` δείχνει στο repository με απόλυτο path. Στο βιβλίο το home σου γράφεται `/home/you`, οπότε το repository είναι στο `/home/you/tfbook/ch16/repos/greeting`. Στο δικό σου μηχάνημα το home είναι άλλο, για παράδειγμα `/Users/maria` σε macOS. Τύπωσε το `source` που ταιριάζει στο δικό σου μηχάνημα.

TERMINAL

```
echo "git::file://$HOME/tfbook/ch16/repos/greeting?ref=v1.0.0"
```

ΕΞΟΔΟΣ

```
git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.0.0
```

Στο δικό σου terminal η γραμμή έχει το δικό σου home. Αυτή τη γραμμή γράφεις στο `source`. Σε όλο το κεφάλαιο, όπου βλέπεις `file:///home/you/`, βάλε το δικό σου home στη θέση του `/home/you`.

main.tf

```
module "greeting" {
  source = "git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.0.0"

  name = "Terraform"
}

output "message" {
  value = module.greeting.message
}
```

Το `source` έχει τρία κομμάτια.

- Το πρόθεμα `git::` λέει ρητά στο Terraform να χρησιμοποιήσει το Git.
- Ακολουθεί η διεύθυνση του repository. Εδώ είναι `file://` με απόλυτο path, γιατί το repository είναι στον δίσκο σου. Οι τρεις κάθετοι είναι οι δύο του `file://` και η πρώτη του path. Σε μια ομάδα η διεύθυνση είναι `https://` ή `ssh://`.
- Το `?ref=v1.0.0` λέει ποια έκδοση θα γίνει checkout. Το `ref` δέχεται tag, κλαδί ή commit.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
Downloading git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.0.0 for greeting...
- greeting in .terraform/modules/greeting

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

TERMINAL

```
ls -a .terraform/modules/greeting
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
.
..
.git
main.tf
outputs.tf
README.md
variables.tf
versions.tf

Changes to Outputs:
  + message = "Γεια σου, Terraform!"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

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

Outputs:

message = "Γεια σου, Terraform!"
```

Η γραμμή `Downloading` γράφει όλο το `source` και ο φάκελος του module έχει `.git`, όπως το module του Registry. Το init έκανε clone το repository και checkout το tag `v1.0.0`. Το module δεν έχει resources, οπότε το apply γράφει μόνο το output.

Μια νέα έκδοση του module

Ο συντηρητής του module προσθέτει ένα input, το `greeting`, με default τον παλιό χαιρετισμό. Όποιος δεν το ορίζει παίρνει το ίδιο αποτέλεσμα με πριν. Προσθέτει επίσης έναν φάκελο `examples` με ένα παράδειγμα χρήσης.

TERMINAL

```
cd ../repos/greeting
```

variables.tf

```
variable "name" {
  type        = string
  description = "Ποιον χαιρετά το module."
}

variable "greeting" {
  type        = string
  default     = "Γεια σου"
  description = "Η λέξη του χαιρετισμού."
}
```

main.tf

```
locals {
  message = "${var.greeting}, ${var.name}!"
}
```

examples/basic/main.tf

```
module "greeting" {
  source = "../../"

  name = "Terraform"
}

output "message" {
  value = module.greeting.message
}
```

TERMINAL

```
git add .
git -c user.name=You -c user.email=you@example.com commit -q -m "Input greeting"
git tag v1.1.0
git tag
```

ΕΞΟΔΟΣ

```
v1.0.0
v1.1.0
```

Το repository έχει τώρα δύο εκδόσεις. Στο `hello` άλλαξε το `ref` σε `v1.1.0` και χρησιμοποίησε το νέο input.

TERMINAL

```
cd ../../hello
```

main.tf

```
module "greeting" {
  source = "git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.1.0"

  name     = "Terraform"
  greeting = "Καλημέρα"
}

output "message" {
  value = module.greeting.message
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Module source has changed
│
│   on main.tf line 2, in module "greeting":
│    2:   source = "git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.1.0"
│
│ The source address was changed since this module was installed. Run
│ "terraform init" to install all modules required by this configuration.
╵
```

Το σφάλμα είναι `Module source has changed`. Για το Terraform το `ref` είναι μέρος του `source`, οπότε μια νέα έκδοση είναι νέα πηγή. Ό,τι υπάρχει στο `.terraform/modules/greeting` είναι ακόμη η v1.0.0. Η λύση είναι και πάλι το `terraform init`.

TERMINAL

```
terraform init
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
Downloading git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.1.0 for greeting...
- greeting in .terraform/modules/greeting

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

```
Changes to Outputs:
  ~ message = "Γεια σου, Terraform!" -> "Καλημέρα, Terraform!"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

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

Outputs:

message = "Καλημέρα, Terraform!"
```

Το init κατέβασε το repository ξανά, αυτή τη φορά στο tag `v1.1.0`. Το plan δείχνει το output να αλλάζει από `"Γεια σου, Terraform!"` σε `"Καλημέρα, Terraform!"`.

Ένα tag δεν αλλάζει από μόνο του. Ένα κλαδί αλλάζει με κάθε commit. Αν γράψεις `?ref=main`, το init κατεβάζει ό,τι έχει το `main` εκείνη τη στιγμή και μετά δεν το ξανακοιτάζει. Ένα νέο commit στο `main` φτάνει στον φάκελό σου μόνο με `terraform init -upgrade`, ενώ στο μηχάνημα ενός συναδέλφου φτάνει με το πρώτο του init. Στην άσκηση 16.5 το βλέπεις να συμβαίνει. Γι' αυτό σε Git καρφώνεις tag ή, για απόλυτη βεβαιότητα, το hash ενός commit.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Σε Git source η έκδοση μπαίνει με `?ref=` στο τέλος της διεύθυνσης, για παράδειγμα `?ref=v1.2.0`. Το argument `version` δεν δουλεύει με Git. Ένα νέο block `module`, μια αλλαγή στο `source` ή ένα `version` που δεν δέχεται πια την εγκατεστημένη έκδοση κάνουν το plan να σταματήσει με σφάλμα. Η λύση είναι πάντα το `terraform init`.

Άλλες μορφές Git και ο υποφάκελος //

Στην πράξη το repository είναι σε κάποιον server. Οι συνηθισμένες μορφές είναι οι παρακάτω.

ΕΞΟΔΟΣ

```
git::https://example.com/network.git?ref=v2.0.0
git::ssh://git@example.com/network.git?ref=v2.0.0
git@github.com:acme/terraform-aws-network.git?ref=v2.0.0
github.com/acme/terraform-aws-network?ref=v2.0.0
```

Οι δύο πρώτες λένε ρητά `git::` και το πρωτόκολλο. Η τρίτη είναι η μορφή SSH που δίνει το GitHub για clone. Η τέταρτη είναι συντόμευση για το GitHub, χωρίς `git::` και χωρίς πρωτόκολλο. Για ιδιωτικά repositories δεν γράφεις κωδικούς στο `source`. Το Terraform τρέχει το `git` του μηχανήματός σου, που χρησιμοποιεί τα SSH keys ή τα credentials που έχεις ήδη ρυθμίσει.

Δες τι κάνει το Terraform με τη συντόμευση. Το repository του `hashicorp/dir/template` είναι δημόσιο, οπότε μπορείς να το καλέσεις απευθείας, παρακάμπτοντας το Registry.

TERMINAL

```
cd ~/tfbook/ch16
mkdir shorthand
cd shorthand
```

main.tf

```
module "docs" {
  source = "github.com/hashicorp/terraform-template-dir?ref=v1.0.2"

  base_dir = path.module
}
```

Πριν από το init τρέξε plan, για να δεις τι λέει το Terraform για ένα module που δεν έχει κατέβει ακόμη.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Module not installed
│
│   on main.tf line 1:
│    1: module "docs" {
│
│ This module is not yet installed. Run "terraform init" to install all
│ modules required by this configuration.
╵
```

Το σφάλμα `Module not installed` εμφανίζεται κάθε φορά που προσθέτεις νέο block `module`. Ζητά το ίδιο με τα προηγούμενα σφάλματα, ένα `terraform init`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...
Downloading git::https://github.com/hashicorp/terraform-template-dir.git?ref=v1.0.2 for docs...
- docs in .terraform/modules/docs

…
```

Η γραμμή `Downloading` δεν γράφει αυτό που έγραψες. Το Terraform μετέτρεψε τη συντόμευση στην πλήρη μορφή, `git::https://github.com/` και `.git` στο τέλος. Είναι ο ίδιος κώδικας με την έκδοση 1.0.2 του Registry. Τώρα όμως το Registry δεν συμμετέχει, οπότε η έκδοση ορίζεται μόνο με `ref` και όχι με `version`.

Ένα repository μπορεί να έχει πολλά modules σε υποφακέλους. Τον υποφάκελο τον γράφεις μετά από διπλή κάθετο, το `//`. Το `?ref` μπαίνει μετά τον υποφάκελο.

ΕΞΟΔΟΣ

```
git::https://example.com/network.git//modules/vpc?ref=v2.0.0
```

Το init κατεβάζει ολόκληρο το repository και χρησιμοποιεί ως module τον φάκελο `modules/vpc`. Αφού κατεβαίνει όλο, ένα module στον υποφάκελο μπορεί να καλεί άλλο module του ίδιου repository με τοπικό path, όπως το `../subnet`.

HTTP, S3 και GCS

Ένα module μπορεί να διανέμεται και ως αρχείο, `.zip` ή `.tar.gz`. Το init το κατεβάζει, το αποσυμπιέζει και το χρησιμοποιεί. Η έκδοση φαίνεται συνήθως στο όνομα του αρχείου.

HCL

```
module "vpc" {
  source = "https://example.com/modules/vpc-1.2.0.zip"
}

module "dns" {
  source = "s3::https://s3-eu-central-1.amazonaws.com/acme-modules/dns-1.0.0.zip"
}

module "cdn" {
  source = "gcs::https://www.googleapis.com/storage/v1/acme-modules/cdn-1.0.0.zip"
}
```

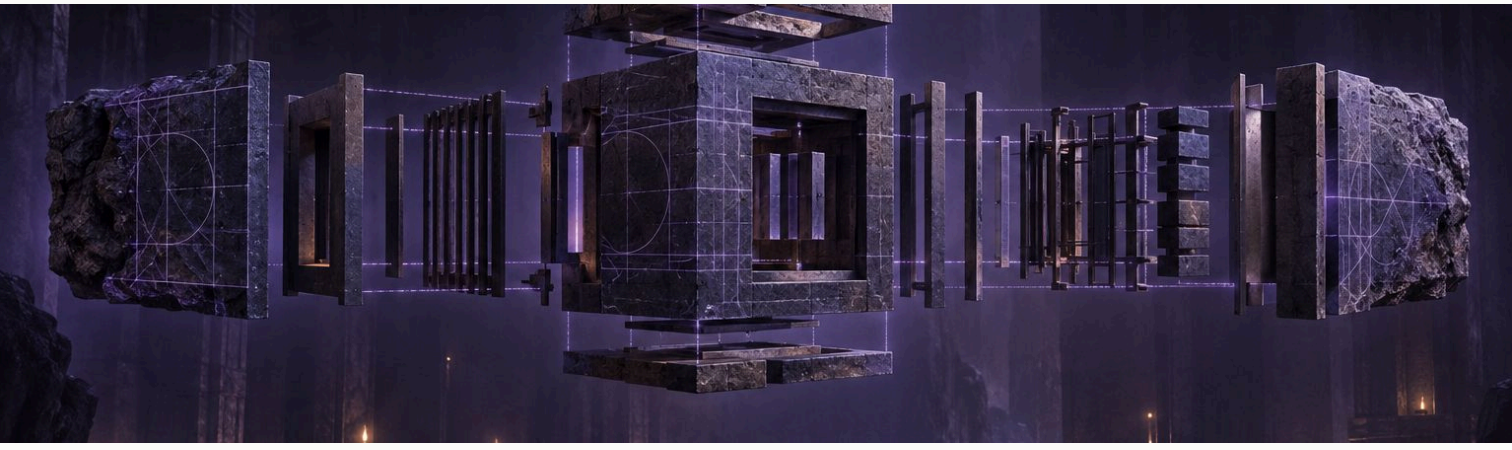
Τα προθέματα `s3::` και `gcs::` λένε στο Terraform να κατεβάσει το αρχείο από bucket του S3 ή του Google Cloud Storage. Για την πρόσβαση χρησιμοποιεί τα credentials του αντίστοιχου cloud που υπάρχουν στο περιβάλλον σου. Τα credentials του AWS τα γνωρίζεις στο κεφάλαιο 17. Αυτές οι πηγές είναι πιο σπάνιες από το Registry και το Git. Για την εξέταση αρκεί να τις αναγνωρίζεις.

Η δομή ενός δημοσιευμένου module

Ένα module που θα χρησιμοποιήσουν άλλοι ακολουθεί μια τυπική δομή αρχείων, το **standard module structure**. Το repository `greeting` την ακολουθεί ήδη. Του λείπει μόνο το `LICENSE`, που χρειάζεται όταν το module γίνει δημόσιο.

TERMINAL

```
git -C ~/tfbook/ch16/repos/greeting ls-files
```



```
ΕΞΟΔΟΣ

README.md
examples/basic/main.tf
main.tf
outputs.tf
variables.tf
versions.tf
```

Αρχείο ή φάκελος	Τι περιέχει
`README.md`	Τι κάνει το module και πώς χρησιμοποιείται
`LICENSE`	Η άδεια με την οποία μπορούν να το χρησιμοποιούν άλλοι
`main.tf`	Τα resources και τα locals, το κυρίως σώμα
`variables.tf`	Τα inputs, το καθένα με `description`
`outputs.tf`	Τα outputs, το καθένα με `description`
`versions.tf`	Το block `terraform` με `required_version` και `required_providers`
`examples/`	Πλήρη παραδείγματα χρήσης, το καθένα ένα root module
`modules/`	Εσωτερικά modules, που καλούνται και μόνα τους με `//`

Τα ονόματα των αρχείων είναι σύμβαση. Το Terraform διαβάζει κάθε `.tf` όποιο όνομα κι αν έχει. Το `hashicorp/dir/template` έχει `files.tf` αντί για `main.tf` και φάκελο `test` αντί για `examples`. Η σύμβαση όμως βοηθά όποιον διαβάζει το module για πρώτη φορά. Το Registry χρησιμοποιεί επίσης το README και τα `description` για να φτιάξει τη σελίδα τεκμηρίωσης κάθε module.

Το παράδειγμα στο `examples/basic` καλεί το module με `source = "../../"`, δηλαδή τη ρίζα του repository. Είναι κανονικό τοπικό path και δουλεύει όπως στο κεφάλαιο 15.

Για να δημοσιεύσεις ένα module στο δημόσιο Registry οι βασικές απαιτήσεις είναι τρεις.

1. Δημόσιο repository στο GitHub με όνομα `terraform-PROVIDER-NAME`, για παράδειγμα `terraform-aws-vpc`.
2. Τη δομή που περιγράψαμε, με τα αρχεία `.tf` στη ρίζα.
3. Tags με semantic version, όπως `v1.0.0` ή `1.0.0`. Κάθε tag γίνεται μία έκδοση.

Μια εταιρεία συνήθως δεν θέλει τα modules της δημόσια. Το HCP Terraform έχει **private registry** για κάθε organization. Τα modules του έχουν το hostname μπροστά, για παράδειγμα `app.terraform.io/acme/vpc/aws`, όπου `acme` είναι το organization. Δέχονται `version` ακριβώς όπως τα modules του δημόσιου Registry. Το init χρειάζεται credentials για να τα κατεβάσει. Το κεφάλαιο 26 δείχνει πώς δημοσιεύεις εκεί και πώς συνδέεσαι με το `terraform login`.

> **ΣΗΜΕΙΩΣΗ**
> Όπως με τους providers, ένα module από άγνωστο namespace το ελέγχεις πριν το χρησιμοποιήσεις. Ο κώδικάς του τρέχει με τα δικά σου credentials. Ένα ακριβές `version` εξασφαλίζει επιπλέον ότι δεν θα αλλάξει κάτω από τα πόδια σου.

Το source είναι σταθερό κείμενο

Το init κατεβάζει τα modules πριν διαβάσει τιμές μεταβλητών και πριν υπολογίσει εκφράσεις. Γι' αυτό το `source` και το `version` πρέπει να είναι σταθερά strings. Δοκίμασε να βάλεις την έκδοση σε μεταβλητή.

TERMINAL

```
cd ~/tfbook/ch16
mkdir literal
cd literal
```

main.tf

```
variable "template_version" {
  type    = string
  default = "1.0.2"
}

module "docs" {
  source  = "hashicorp/dir/template"
  version = var.template_version

  base_dir = path.module
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Initializing modules...

╷
│ Error: Unknown module version
│
│   on main.tf line 8, in module "docs":
│    8:   version = var.template_version
│
│ Only literal values and const variables can be evaluated during init.
╵
```

Το σφάλμα `Unknown module version` λέει ότι στο init υπολογίζονται μόνο σταθερές τιμές και μεταβλητές `const`. Το ίδιο ισχύει για το `source`. Ο κανόνας για την εξέταση είναι απλός. Το `source` και το `version` γράφονται ως σταθερά strings.

ΣΗΜΕΙΩΣΗ

Από το Terraform 1.15 το `source` και το `version` δέχονται locals με σταθερή τιμή και input variables που δηλώνονται με `const = true`. Αυτό είναι το `const` του μηνύματος. Είναι εκτός ύλης εξέτασης. Η εξέταση βασίζεται στην 1.12, όπου το `source` και το `version` πρέπει να είναι σταθερά strings. Εκεί η ίδια configuration σταματά με `Variables not allowed`.

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch16`. Όσες χρησιμοποιούν το repository `greeting` το βρίσκουν στο `~/tfbook/ch16/repos/greeting`, με τα tags `v1.0.0` και `v1.1.0` του κεφαλαίου. Στις εκφωνήσεις και στις λύσεις το `source` γράφει `file:///home/you/`. Βάλε στη θέση του `/home/you` το δικό σου home, όπως στο κεφάλαιο.

ΑΣΚΗΣΗ 16.1 version σε Git

Κάποιος συνάδελφος έγραψε την παρακάτω configuration. Τρέξε `terraform init` και διάβασε το σφάλμα. Διόρθωσέ την ώστε να φέρνει την έκδοση v1.1.0 του module και κάνε apply.

main.tf

```
module "greeting" {
  source  = "git::file:///home/you/tfbook/ch16/repos/greeting"
  version = "1.1.0"

  name = "Git"
}

output "message" {
  value = module.greeting.message
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
╷
│ Error: Invalid registry module source address
…
│ Failed to parse module registry address: a module registry source address
│ must have either three or four slash-separated components.
│
│ Terraform assumed that you intended a module registry source address
│ because you also set the argument "version", which applies only to registry
│ modules.
…
```

ΑΣΚΗΣΗ 16.2 Templates με κατάληξη .tftpl

Στο κεφάλαιο 10 τα templates είχαν κατάληξη `.tftpl`. Φτιάξε φάκελο `pages` με δύο αρχεία. Το template `index.html.tftpl` περιέχει `<p>Υπεύθυνη ${owner}</p>`. Το στατικό `robots.txt` περιέχει `User-agent: *`. Κάλεσε το `hashicorp/dir/template` 1.0.2 ώστε να θεωρεί templates τα `.tftpl`, με `owner` την `Ελένη`. Γράψε ένα output `rendered` με το `content` του `index.html` από το `files_in_memory` χωρίς την αλλαγή γραμμής του τέλους. Γράψε και ένα output `static` με τα κλειδιά του `files_on_disk`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
rendered = "<p>Υπεύθυνη Ελένη</p>"
static = [
  "robots.txt",
]
```

ΑΣΚΗΣΗ 16.3 Μια έκδοση που δεν υπάρχει

Ένα README λέει να καλέσεις το `hashicorp/dir/template` με `version = "~> 2.0"` και `base_dir = path.module`. Τρέξε `terraform init`. Ποια είναι η νεότερη έκδοση σύμφωνα με το μήνυμα; Γράψε μετά ένα constraint που δέχεται μόνο patches της σειράς 1.0 και ξανατρέξε το init.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
│ Error: Unresolvable module version constraint
…
│ There is no available version of module
│ "registry.terraform.io/hashicorp/dir/template" (main.tf:1) which matches
│ the given version constraint. The newest available version is 1.0.2.
…
```

ΑΣΚΗΣΗ 16.4 Νέο input σε παλιά έκδοση

Γράψε configuration που καλεί το module `greeting` στην έκδοση v1.0.0 με `name = "Terraform"` και `greeting = "Καλησπέρα"`. Τρέξε `terraform init`. Διόρθωσε το πρόβλημα χωρίς να σβήσεις το `greeting` και κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
message = "Καλησπέρα, Terraform!"
```

ΑΣΚΗΣΗ 16.5 Κλαδί αντί για tag

Κάλεσε το module `greeting` με `?ref=main` και `name = "Terraform"`, κάνε init και apply. Στο repository άλλαξε το `default` του `greeting` σε `Χαίρε` και κάνε commit χωρίς tag. Τρέξε `terraform init` και `terraform plan`. Μετά τρέξε `terraform init -upgrade` και ξανά `terraform plan`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Changes to Outputs:
  ~ message = "Γεια σου, Terraform!" -> "Χαίρε, Terraform!"
…
```

ΑΣΚΗΣΗ 16.6 Module σε υποφάκελο

Πρόσθεσε στο repository `greeting` ένα εσωτερικό module στο `modules/shout/main.tf`. Δέχεται ένα input `text` και έχει output `loud` με το `upper(var.text)`. Κάνε commit και βάλε tag `v1.2.0`. Σε νέο φάκελο κάλεσέ το με `text = "γεια σου"`, χρησιμοποιώντας `//` και `?ref=v1.2.0`. Τι γράφει το init για το πού μπήκε το module;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Downloading git::file:///home/you/tfbook/ch16/repos/greeting?ref=v1.2.0 for shout...
- shout in .terraform/modules/shout/modules/shout
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 16.1

Ποια από τις παρακάτω πηγές modules δέχεται το argument `version`;

a. Τοπικό path όπως `./modules/network`
b. Git repository όπως `git::https://example.com/network.git`
c. Terraform Registry όπως `acme/network/aws`
d. Αρχείο HTTP όπως `https://example.com/network.zip`

ΕΡΩΤΗΣΗ 16.2

Το module σου βρίσκεται στον υποφάκελο `modules/network` του φακέλου εργασίας. Ποια τιμή του `source` το φορτώνει;

a. `"modules/network"`
b. `"./modules/network"`
c. `"local::modules/network"`
d. `"file://modules/network"`

ΕΡΩΤΗΣΗ 16.3

Θέλεις το module από το repository `https://example.com/network.git` στο tag `v1.2.0`. Ποιο `source` είναι σωστό;

a. `"git::https://example.com/network.git"` μαζί με `version = "v1.2.0"`
b. `"git::https://example.com/network.git@v1.2.0"`
c. `"git::https://example.com/network.git//v1.2.0"`
d. `"git::https://example.com/network.git?ref=v1.2.0"`

ΕΡΩΤΗΣΗ 16.4

Σωστό ή λάθος; Το `.terraform.lock.hcl` καταγράφει την έκδοση κάθε module του Registry που επέλεξε το init, ώστε όλη η ομάδα να παίρνει την ίδια.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 16.5

Άλλαξες το `version` ενός module του Registry από `"1.0.1"` σε `"1.0.2"`. Το `terraform plan` σταματά με `Module version requirements have changed`. Τι πρέπει να τρέξεις;

a. `terraform init`
b. `terraform validate`
c. `terraform apply -auto-approve`
d. `terraform fmt`

ΕΡΩΤΗΣΗ 16.6

Ποια ισχύουν για ένα module που δημοσιεύεται στο δημόσιο Terraform Registry; Επίλεξε δύο.

a. Βρίσκεται σε δημόσιο repository στο GitHub με όνομα της μορφής `terraform-PROVIDER-NAME`.
b. Το Registry κρατά και το state όσων το χρησιμοποιούν.
c. Κάθε νέα έκδοση είναι ένα Git tag με semantic version, όπως `v1.3.0`.
d. Η διεύθυνσή του γράφεται `NAMESPACE/PROVIDER/NAME`, με τον provider στη μέση.



ΜΕΡΟΣ IV

Terraform στο AWS

Ξέρεις πλέον τη γλώσσα, το state και τα modules. Από εδώ και πέρα ο κώδικας μιλά με το API του AWS και φτιάχνει buckets, δίκτυα και servers. Τα παραδείγματα τρέχουν σε έναν εξομοιωτή του AWS στον υπολογιστή σου ή σε λογαριασμό δοκιμών. Ο κύκλος write, plan, apply μένει ίδιος. Αλλάζει μόνο ότι κάθε apply αγγίζει πλέον ένα πραγματικό API.



ΚΕΦΑΛΑΙΟ 17

Ο AWS provider

Ο `hashicorp/aws` μεταφράζει τα resources σου σε κλήσεις στο API του AWS. Σε αυτό το κεφάλαιο στήνεις το περιβάλλον όπου τρέχουν τα παραδείγματα του AWS, μαθαίνεις πού βρίσκει ο provider credentials και region και φτιάχνεις το πρώτο σου S3 bucket. Μετά βάζεις buckets σε δύο regions με aliases και δένεις τον AWS με άλλους providers στο ίδιο workflow.

Πού τρέχουν τα παραδείγματα

Τα κεφάλαια 17 έως 25 και το 27 χρειάζονται ένα API που μιλά σαν το AWS. Έχεις δύο τρόπους να το αποκτήσεις. Διάλεξε έναν και κράτα τον σε όλο το υπόλοιπο βιβλίο.

Ο πρώτος τρόπος, Moto

Το **Moto** είναι εξομοιωτής του AWS API, ανοιχτού κώδικα. Δέχεται τις ίδιες αιτήσεις με το AWS για το S3, το EC2, το IAM, το STS και δεκάδες ακόμη υπηρεσίες και κρατά ό,τι φτιάχνεις στη μνήμη του. Δεν χρειάζεσαι λογαριασμό ούτε πληρώνεις. Με αυτό εκτελέστηκε το βιβλίο. Το ξεκινάς σε Docker.

TERMINAL

```
docker run -d --name moto -p 5050:5000 motoserver/moto:5.2.3
```

Το `-p 5050:5000` συνδέει τη θύρα 5050 του υπολογιστή σου με τη θύρα 5000 του container, όπου ακούει το Moto. Μετά λες στο τερματικό πού είναι το API και ποια κλειδιά θα στέλνεις.

TERMINAL

```
export AWS_ENDPOINT_URL=http://localhost:5050
export AWS_ACCESS_KEY_ID=test
export AWS_SECRET_ACCESS_KEY=test
export AWS_REGION=eu-central-1
```

Το Moto δέχεται οποιαδήποτε κλειδιά. Οι μεταβλητές ισχύουν μόνο στο τερματικό όπου τις έγραψες. Το Moto κρατά τα resources όσο ζει το container, ενώ τα state των φακέλων σου τα θυμούνται και μετά. Γι' αυτό κάνε `terraform destroy` πριν το σβήσεις με `docker rm -f moto`.

Ο δεύτερος τρόπος, λογαριασμός sandbox

Ο δεύτερος τρόπος είναι ένας πραγματικός λογαριασμός AWS μόνο για δοκιμές, χωριστός από κάθε λογαριασμό με δεδομένα που μετράνε. Η σύνδεση γίνεται συνήθως με το IAM Identity Center, την υπηρεσία SSO του AWS. Το `aws configure sso` ρωτά τη διεύθυνση εισόδου του οργανισμού σου, ανοίγει τον browser και γράφει ένα profile με το όνομα που διαλέγεις, εδώ `tfbook`. Αν ο λογαριασμός είναι δικός σου και δεν έχει ακόμη SSO, ενεργοποίησε πρώτα το IAM Identity Center στην κονσόλα. Φτιάξε εκεί έναν χρήστη και δώσ' του πρόσβαση στον λογαριασμό με το permission set `AdministratorAccess`. Η διεύθυνση εισόδου είναι το AWS access portal URL που δείχνει η αρχική σελίδα του Identity Center.

TERMINAL

```
aws configure sso
```

Το profile στο `~/.aws/config` μοιάζει με το παρακάτω.

~/.aws/config

```
[profile tfbook]
sso_session = tfbook
sso_account_id = 111122223333
sso_role_name = AdministratorAccess
region = eu-central-1

[sso-session tfbook]
sso_start_url = https://example.awsapps.com/start
sso_region = eu-central-1
sso_registration_scopes = sso:account:access
```

Στην αρχή κάθε συνεδρίας δουλειάς κάνεις login και διαλέγεις το profile.

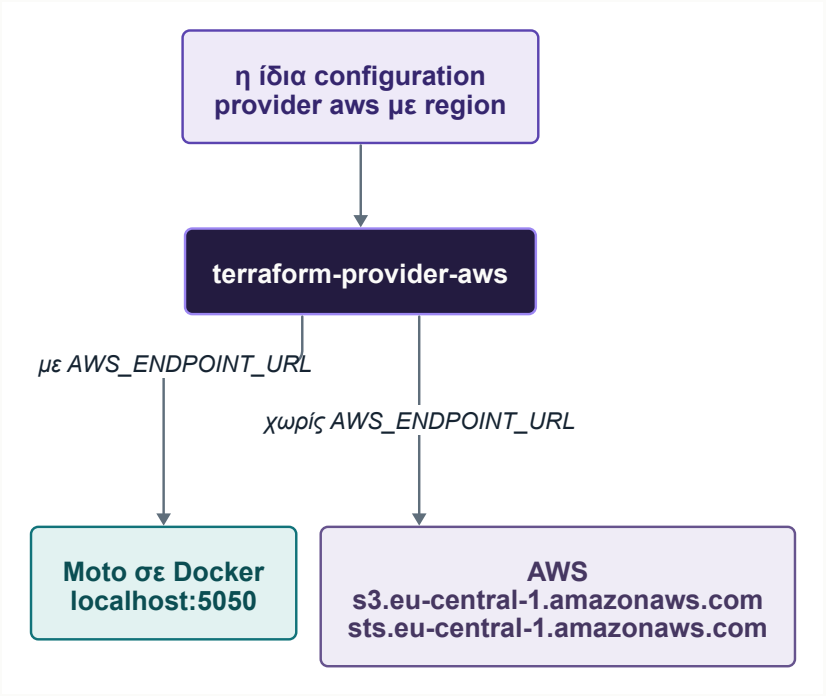
TERMINAL

```
aws sso login --profile tfbook
export AWS_PROFILE=tfbook
unset AWS_ENDPOINT_URL AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY
```

Το `aws sso login` κρατά ένα token που λήγει σε λίγες ώρες. Με αυτό το CLI και ο provider παίρνουν προσωρινά credentials. Το `AWS_PROFILE` διαλέγει το profile για το CLI και για τον provider. Το `unset` σβήνει τις μεταβλητές του Moto. Χωρίς αυτό οι αιτήσεις θα πήγαιναν ακόμη στο Moto και τα κλειδιά `test` θα κέρδιζαν το profile, όπως θα δεις στη σειρά των credentials.

Γιατί ο ίδιος κώδικας δουλεύει και στα δύο

Ο provider και το `aws` CLI βρίσκουν μόνοι τους τη διεύθυνση κάθε υπηρεσίας από το region, για παράδειγμα `s3.eu-central-1.amazonaws.com` για το S3 στη Φρανκφούρτη. Όταν υπάρχει η `AWS_ENDPOINT_URL`, κάθε αίτηση πηγαίνει στη διεύθυνση που γράφει και το Moto απαντά για όλες τις υπηρεσίες από μία θύρα. Η configuration δεν αναφέρει πουθενά το Moto. Ο προορισμός ζει στο περιβάλλον του τερματικού, οπότε τα ίδια αρχεία `.tf` δουλεύουν στο Moto και στο AWS χωρίς καμία αλλαγή.



Σχήμα 17.1 · Η ίδια configuration πηγαίνει στο Moto ή στο AWS. Τον προορισμό τον διαλέγει η μεταβλητή περιβάλλοντος AWS_ENDPOINT_URL και όχι ο κώδικας.

Στο Moto ο λογαριασμός είναι πάντα ο `123456789012`. Τα IDs που δίνει μοιάζουν με του AWS αλλά αλλάζουν σε κάθε εκτέλεση, οπότε η δική σου έξοδος θα έχει άλλα. Το κείμενο δεν βασίζεται ποτέ σε συγκεκριμένο ID.

Έλεγξε ότι μιλάς με το API που περιμένεις.

TERMINAL

```
aws sts get-caller-identity
```

ΕΞΟΔΟΣ

```
{
    "UserId": "AKIAIOSFODNN7EXAMPLE",
    "Account": "123456789012",
    "Arn": "arn:aws:sts::123456789012:user/moto"
}
```

Το STS είναι η υπηρεσία που εκδίδει credentials. Η κλήση `GetCallerIdentity` απαντά στο «ποιος είμαι» χωρίς να χρειάζεται καμία άδεια. Το `Arn` (Amazon Resource Name) είναι το πλήρες όνομα της

ταυτότητας που κάνει τις αιτήσεις, εδώ ο χρήστης `moto`. Με SSO θα δεις τον δικό σου λογαριασμό και ένα ARN με `assumed-role`. Κράτα τον λογαριασμό σε μια μεταβλητή του shell για τις εντολές ελέγχου.

TERMINAL

```
ACCOUNT=$(aws sts get-caller-identity --query Account --output text)
echo "$ACCOUNT"
```

ΕΞΟΔΟΣ

```
123456789012
```

Το `--query` κρατά ένα πεδίο της απάντησης και το `--output text` το τυπώνει χωρίς εισαγωγικά.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Ένας λογαριασμός sandbox θέλει τρεις προφυλάξεις από την πρώτη μέρα. Ένα budget στο Billing με ειδοποίηση σε email, ώστε να μάθεις αμέσως αν κάτι ξεχάστηκε ανοιχτό. Καμία δουλειά με τον root user. Και `terraform destroy` στο τέλος κάθε κεφαλαίου, γιατί ο server του κεφαλαίου 18 χρεώνεται για κάθε ώρα που τρέχει.

Ποιος είμαι, κατά το Terraform

Φτιάξε τον φάκελο του πρώτου παραδείγματος.

TERMINAL

```
mkdir first
cd first
```

Το `main.tf` δηλώνει τον provider και τον ρυθμίζει.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}
```

Το block `provider` έχει μία ρύθμιση, το `region`. Το AWS είναι χωρισμένο σε **regions**, περιοχές του κόσμου με δικά τους datacenters. Το `eu-central-1` είναι η Φρανκφούρτη και το `eu-west-1` η Ιρλανδία. Κάθε resource ζει σε ένα region. Πριν φτιάξεις οτιδήποτε, ρώτα τον provider ποιος είσαι με δύο data sources του κεφαλαίου 12.

identity.tf

```
data "aws_caller_identity" "current" {}

data "aws_region" "current" {}

output "account_id" {
  value = data.aws_caller_identity.current.account_id
}

output "caller_arn" {
  value = data.aws_caller_identity.current.arn
}

output "region" {
  value = data.aws_region.current.region
}
```

Το `aws_caller_identity` κάνει την ίδια κλήση `GetCallerIdentity` με το CLI και δίνει τα attributes `account_id`, `arn` και `user_id`. Το `aws_region` λέει σε ποιο region δουλεύει ο provider, στο attribute `region`. Κανένα δεν θέλει arguments. Το όνομα `current` είναι σύμβαση για data sources που περιγράφουν το τρέχον περιβάλλον.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Finding hashicorp/aws versions matching "6.67.0"...
- Installing hashicorp/aws v6.67.0...
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

Ο `hashicorp/aws` είναι ο μεγαλύτερος provider του βιβλίου. Το πακέτο είναι περίπου 180 MB και το εκτελέσιμο στο `.terraform` πιάνει περίπου 750 MB, οπότε το πρώτο init θέλει λίγη υπομονή.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.aws_region.current: Reading...
data.aws_caller_identity.current: Reading...
data.aws_region.current: Read complete after 0s [id=eu-central-1]
data.aws_caller_identity.current: Read complete after 0s [id=123456789012]

Changes to Outputs:
  + account_id = "123456789012"
  + caller_arn = "arn:aws:sts::123456789012:user/moto"
  + region     = "eu-central-1"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

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

Outputs:

account_id = "123456789012"
caller_arn = "arn:aws:sts::123456789012:user/moto"
region = "eu-central-1"
```

Δεν υπάρχει κανένα resource, γι' αυτό η περίληψη γράφει `0 added`. Ο λογαριασμός και το ARN είναι ίδια με του CLI και το region είναι αυτό του block `provider`. Ο provider βρήκε credentials και το API απαντά.

> **ΠΑΓΙΔΑ**
> Σε παλιότερα παραδείγματα θα δεις `data.aws_region.current.name`. Στον provider 6.67.0 το `name` είναι deprecated και τη θέση του έχει πάρει το `region`. Δουλεύει ακόμη, αλλά κάθε plan τυπώνει `Warning: Deprecated value used` με το μήνυμα `name is deprecated. Use region instead.` Η άσκηση 17.1 το δείχνει.

Ένα cache για τον provider

Κάθε νέος φάκελος θέλει το δικό του init. Για να μην κατεβαίνουν κάθε φορά 180 MB, ρύθμισε ένα **plugin cache**, έναν φάκελο όπου το init κρατά κάθε provider που κατεβάζει. Βάλε τη γραμμή `export` στο `~/.bashrc` ή στο `~/.zshrc`.

TERMINAL

```
mkdir -p ~/.terraform.d/plugin-cache
export TF_PLUGIN_CACHE_DIR=~/.terraform.d/plugin-cache
```

Από εκεί και πέρα το init φτιάχνει σε κάθε φάκελο μόνο έναν σύνδεσμο προς το cache, με έναν όρο. Ο φάκελος πρέπει να έχει ήδη lock file με τον provider, ώστε το init να ξέρει ποιο checksum περιμένει. Χωρίς lock file τον κατεβάζει ξανά. Γι' αυτό σε κάθε νέο φάκελο του κεφαλαίου αντιγράφουμε πρώτα το lock file του `first`.

Το πρώτο bucket

Το **S3** (Simple Storage Service) αποθηκεύει αρχεία, που εδώ λέγονται **objects**. Κάθε object ζει σε ένα **bucket** και έχει ένα **key**, το όνομά του μέσα στο bucket. Το bucket ανήκει σε ένα region, αλλά το όνομά του είναι μοναδικό σε όλο το AWS. Αν οποιοσδήποτε λογαριασμός στον κόσμο έχει ήδη bucket

με το όνομα που ζητάς, η δημιουργία αποτυγχάνει με `BucketAlreadyExists`. Το όνομα έχει 3 έως 63 χαρακτήρες από πεζά λατινικά, ψηφία, τελείες και παύλες. Ξεκινά και τελειώνει με γράμμα ή ψηφίο.

Στο Moto κάθε όνομα είναι ελεύθερο. Για να δουλεύει ο ίδιος κώδικας και στο AWS, βάλε στο όνομα τον αριθμό του λογαριασμού, που δεν τον έχει κανείς άλλος.

bucket.tf

```
resource "aws_s3_bucket" "notes" {
  bucket = "tfbook-${data.aws_caller_identity.current.account_id}-notes"
}
```

Το argument `bucket` είναι το όνομα, εδώ `tfbook-123456789012-notes`. Τα plans του AWS είναι μεγάλα, γιατί κάθε resource έχει δεκάδες attributes. Από εδώ και πέρα το βιβλίο κρατά μόνο τις γραμμές που εξηγεί.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
data.aws_region.current: Read complete after 0s [id=eu-central-1]
data.aws_caller_identity.current: Read complete after 0s [id=123456789012]
…
  # aws_s3_bucket.notes will be created
  + resource "aws_s3_bucket" "notes" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "tfbook-123456789012-notes"
…
      + force_destroy               = false
…
      + id                          = (known after apply)
…
      + region                      = "eu-central-1"
…
    }
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
```

Το data source διαβάστηκε μέσα στο plan, οπότε το `bucket` είναι ήδη γνωστό. Το `force_destroy = false` είναι προεπιλογή του provider και θα το συναντήσεις στο καθάρισμα. Το `region` το πήρε το resource από το block `provider`. Τα περισσότερα attributes, όπως το `arn` και το `id`, γράφουν `(known after apply)` γιατί τα δίνει το AWS μετά τη δημιουργία.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.notes: Creating...
aws_s3_bucket.notes: Creation complete after 1s [id=tfbook-123456789012-notes]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
```

TERMINAL

```
aws s3 ls
```

ΕΞΟΔΟΣ

```
2026-10-08 05:29:02 tfbook-123456789012-notes
```

Το `aws s3 ls` δείχνει τα buckets του λογαριασμού με την ημερομηνία δημιουργίας τους. Το `id` του bucket, στις αγκύλες του apply, είναι το όνομά του.

Ένα object στο bucket

Πρόσθεσε ένα αρχείο κειμένου στο `bucket.tf`.

bucket.tf

```
resource "aws_s3_bucket" "notes" {
  bucket = "tfbook-${data.aws_caller_identity.current.account_id}-notes"
}

resource "aws_s3_object" "hello" {
  bucket       = aws_s3_bucket.notes.id
  key          = "hello.txt"
  content      = "Γεια σου, S3\n"
  content_type = "text/plain; charset=utf-8"
}
```

Η αναφορά στο `id` του bucket φτιάχνει implicit dependency, όπως στο κεφάλαιο 6, οπότε το object περιμένει το bucket. Το `key` είναι το όνομα του object. Το `content` δίνει το περιεχόμενο μέσα στον κώδικα, ενώ για αρχείο του δίσκου γράφεις `source` με το path του. Το `content_type` είναι ο τύπος που στέλνει το S3 σε όποιον κατεβάζει το object.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.notes: Refreshing state... [id=tfbook-123456789012-notes]
…
aws_s3_object.hello: Creating...
aws_s3_object.hello: Creation complete after 0s [id=tfbook-123456789012-notes/hello.txt]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
```

TERMINAL

```
aws s3 ls "s3://tfbook-$ACCOUNT-notes"
aws s3 cp "s3://tfbook-$ACCOUNT-notes/hello.txt" -
```

ΕΞΟΔΟΣ

```
2026-10-08 05:29:06         20 hello.txt
Γεια σου, S3
```

Με διεύθυνση `s3://` το `aws s3 ls` δείχνει τα objects ενός bucket. Το `hello.txt` έχει 20 bytes, γιατί κάθε ελληνικό γράμμα πιάνει δύο bytes στο UTF-8. Το `aws s3 cp` με προορισμό `-` τυπώνει το περιεχόμενο στο τερματικό.

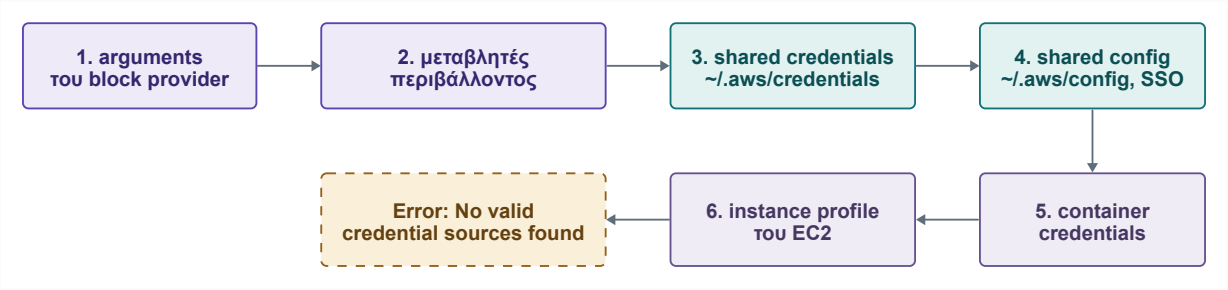
Πού βρίσκει credentials ο provider

Πριν από την πρώτη κλήση ο provider χρειάζεται credentials, για να υπογράψει κάθε αίτηση. Τα ψάχνει με συγκεκριμένη σειρά και σταματά στην πρώτη πηγή που του δίνει.

1. **Arguments του block `provider`.** Τα `access_key` και `secret_key` γράφουν κλειδιά μέσα στον κώδικα. Το `profile` διαλέγει profile από τα αρχεία του βήματος 3 και 4. Το block `assume_role` ζητά προσωρινά credentials για έναν IAM role με βάση όποια credentials βρεθούν.
2. **Μεταβλητές περιβάλλοντος.** Οι `AWS_ACCESS_KEY_ID` και `AWS_SECRET_ACCESS_KEY`, μαζί με την `AWS_SESSION_TOKEN` για προσωρινά credentials. Έτσι δουλεύει το Moto του βιβλίου.



3. **Shared credentials file.** Το `~/.aws/credentials`, με ένα ή περισσότερα profiles. Το profile το διαλέγει η μεταβλητή `AWS_PROFILE`. Χωρίς αυτήν ισχύει το profile `default`.
4. **Shared config file.** Το `~/.aws/config`. Εκεί ζουν τα profiles του SSO. Ο provider χρησιμοποιεί το token που αποθήκευσε το `aws sso login` για να πάρει προσωρινά credentials.
5. **Container credentials.** Όταν το Terraform τρέχει σε container του ECS ή στο CodeBuild, παίρνει τα credentials του role της εργασίας από ένα τοπικό endpoint.
6. **Instance profile.** Όταν τρέχει σε server του EC2 με IAM role, ρωτά την υπηρεσία metadata του server στη διεύθυνση `169.254.169.254`.



Σχήμα 17.2 · Η σειρά με την οποία ο AWS provider ψάχνει credentials. Η πρώτη πηγή που δίνει credentials κερδίζει. Αν καμία δεν δώσει, το plan σταματά με σφάλμα.

Δες την αλυσίδα να εξαντλείται. Το `env -u` τρέχει μια εντολή χωρίς τις μεταβλητές που του λες, εδώ τα κλειδιά του Moto. Αν έχεις ήδη αρχεία στο `~/.aws`, ο provider θα βρει εκεί credentials και η εντολή μπορεί να πετύχει.

TERMINAL

```
env -u AWS_ACCESS_KEY_ID -u AWS_SECRET_ACCESS_KEY terraform plan
```

ΕΞΟΔΟΣ

```
…
╷ Error: No valid credential sources found
…
╷ Error: failed to refresh cached credentials, no EC2 IMDS role found,
…
```

Ο provider δεν βρήκε τίποτα στα βήματα 1 έως 5. Στο βήμα 6 ρώτησε την υπηρεσία metadata του EC2 (IMDS). Δεν πήρε απάντηση, γιατί ο υπολογιστής σου δεν είναι server του EC2. Το `No valid credential sources found` είναι το σφάλμα που βλέπεις κάθε φορά που λείπουν credentials.

Τώρα δώσε credentials με profile. Για να μην αγγίξεις το δικό σου `~/.aws/credentials`, γράψε ένα αρχείο στον φάκελο και δείξ' το με τη μεταβλητή `AWS_SHARED_CREDENTIALS_FILE`. Τα κλειδιά `test` τα δέχεται μόνο το Moto, οπότε σε sandbox λογαριασμό αρκεί να διαβάσεις το παράδειγμα.

credentials

```
[tfbook]
aws_access_key_id     = test
aws_secret_access_key = test
```

TERMINAL

```
env -u AWS_ACCESS_KEY_ID -u AWS_SECRET_ACCESS_KEY \
  AWS_SHARED_CREDENTIALS_FILE=credentials AWS_PROFILE=tfbook \
  terraform plan
rm credentials
```

ΕΞΟΔΟΣ

```
data.aws_caller_identity.current: Reading...
…
data.aws_caller_identity.current: Read complete after 0s [id=123456789012]
…
No changes. Your infrastructure matches the configuration.
…
```

Χωρίς κλειδιά στο περιβάλλον ο provider έφτασε στο βήμα 3, βρήκε το profile `tfbook` και διάβασε τον λογαριασμό. Το `rm` σβήνει το αρχείο, γιατί κλειδιά δεν έχουν θέση δίπλα στον κώδικα. Αν ορίσεις μαζί `AWS_PROFILE` και `AWS_ACCESS_KEY_ID`, τα κλειδιά κερδίζουν, γιατί είναι στο βήμα 2. Γι' αυτό ο sandbox λογαριασμός θέλει το `unset` της αρχής.

ΠΑΓΙΔΑ

Μη γράφεις ποτέ `access_key` και `secret_key` στο block `provider`. Ο κώδικας μπαίνει σε Git και όποιος έχει πρόσβαση στο repository έχει και το κλειδί, για πάντα, ακόμη κι αν το σβήσεις σε επόμενο commit. Στο laptop σου χρησιμοποίησε SSO με profile. Στο CI χρησιμοποίησε role με προσωρινά credentials, για παράδειγμα μέσω OIDC, ή έστω μεταβλητές περιβάλλοντος από τα secrets του CI.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά πού πρέπει να ζουν τα credentials ενός provider. Ποτέ μέσα στα αρχεία `.tf`. Μεταβλητές περιβάλλοντος, profiles, SSO και roles κρατούν τα μυστικά έξω από τον κώδικα και έξω από το version control.

Πού βρίσκει το region

Το region ακολουθεί δική του σειρά.

1. Το argument `region` του block `provider`.
2. Η μεταβλητή `AWS_REGION`.
3. Η μεταβλητή `AWS_DEFAULT_REGION`.
4. Το `region` του profile στο `~/.aws/config`.
5. Σε server του EC2, το region του server από την υπηρεσία metadata, όταν από εκεί ήρθαν και τα credentials.

Αν το `main.tf` γράφει `eu-central-1`, ένα `AWS_REGION=eu-west-1` στο τερματικό δεν αλλάζει τίποτα. Αν ο provider δεν βρει region πουθενά, το plan σταματά με το σφάλμα `invalid AWS Region`. Το βιβλίο γράφει πάντα `region` στο block `provider`, ώστε η configuration να λέει μόνη της πού ζουν τα resources της. Το `AWS_REGION` του Moto το χρειάζεται κυρίως το `aws` CLI.

default_tags και tags_all

Τα **tags** είναι ζεύγη κλειδιού και τιμής πάνω στα resources του AWS. Τα χρησιμοποιούν οι αναφορές κόστους, η αναζήτηση και τα δικαιώματα. Τα περισσότερα resources έχουν argument `tags`. Όσα tags

θέλεις παντού, όπως το project και την ομάδα, τα γράφεις μία φορά στο block `default_tags` του provider.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"

  default_tags {
    tags = {
      Project = "tfbook"
      Owner   = "platform"
    }
  }
}
```

Το bucket παίρνει και δικά του tags. Το `Owner` υπάρχει και στα `default_tags` με άλλη τιμή.

bucket.tf

```
resource "aws_s3_bucket" "notes" {
  bucket = "tfbook-${data.aws_caller_identity.current.account_id}-notes"

  tags = {
    Name  = "notes"
    Owner = "you"
  }
}

resource "aws_s3_object" "hello" {
  bucket       = aws_s3_bucket.notes.id
  key          = "hello.txt"
  content      = "Γεια σου, S3\n"
  content_type = "text/plain; charset=utf-8"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.notes will be updated in-place
  ~ resource "aws_s3_bucket" "notes" {
        id                          = "tfbook-123456789012-notes"
      ~ tags                        = {
          + "Name"  = "notes"
          + "Owner" = "you"
```

```
        }
      ~ tags_all                    = {
          + "Name"    = "notes"
          + "Owner"   = "you"
          + "Project" = "tfbook"
        }
        # (14 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

  # aws_s3_object.hello will be updated in-place
  ~ resource "aws_s3_object" "hello" {
        id                            = "tfbook-123456789012-notes/hello.txt"
        tags                          = {}
      ~ tags_all                      = {
          + "Owner"   = "platform"
          + "Project" = "tfbook"
        }
        # (26 unchanged attributes hidden)
    }
…
Plan: 0 to add, 2 to change, 0 to destroy.
…
```

Το `tags` κρατά μόνο όσα έγραψες στο resource. Το `tags_all` είναι η ένωση των `default_tags` με τα `tags`, δηλαδή όσα θα υπάρχουν πραγματικά στο AWS. Στο bucket το `Owner` έχει τιμή `you`, γιατί η τιμή του resource κερδίζει την προεπιλογή του provider. Το object δεν έχει δικά του tags, οπότε το `tags` μένει `{}` και το `tags_all` παίρνει μόνο τα δύο `default_tags`. Και οι δύο αλλαγές είναι update in place.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

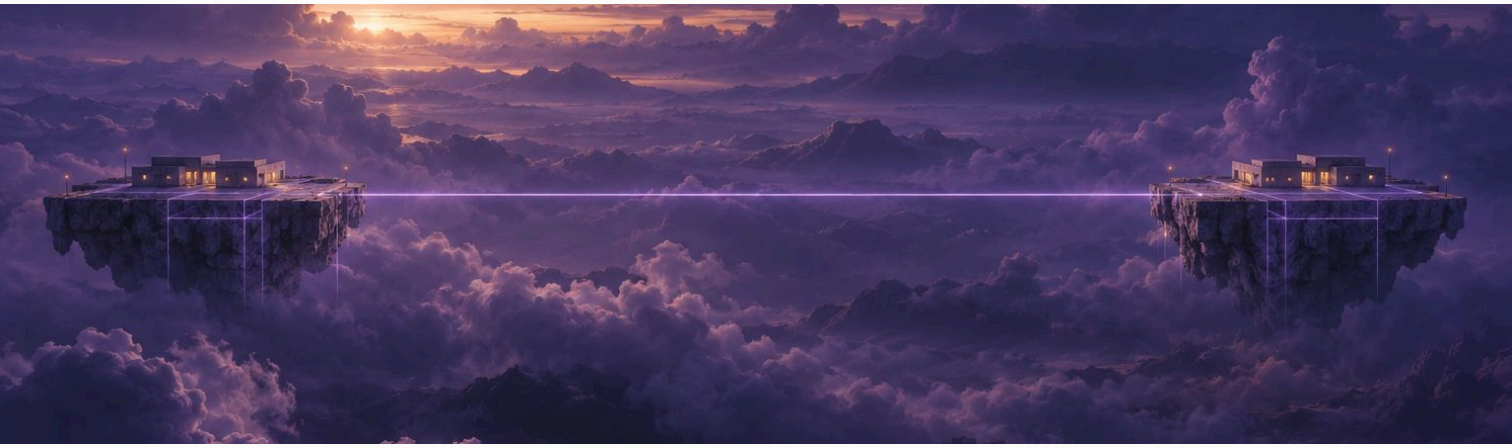
```
…
aws_s3_bucket.notes: Refreshing state... [id=tfbook-123456789012-notes]
aws_s3_object.hello: Refreshing state... [id=tfbook-123456789012-notes/hello.txt]
…
aws_s3_bucket.notes: Modifying... [id=tfbook-123456789012-notes]
aws_s3_bucket.notes: Modifications complete after 0s [id=tfbook-123456789012-notes]
aws_s3_object.hello: Modifying... [id=tfbook-123456789012-notes/hello.txt]
aws_s3_object.hello: Modifications complete after 0s [id=tfbook-123456789012-notes/hello.txt]
…
Apply complete! Resources: 0 added, 2 changed, 0 destroyed.
…
```

TERMINAL

```
aws s3api get-bucket-tagging --bucket "tfbook-$ACCOUNT-notes" \
  --query 'sort_by(TagSet,&Key)[].[Key,Value]' --output text
```

ΕΞΟΔΟΣ

```
Name    notes
Owner   you
Project tfbook
```



Το `aws s3api` δίνει πρόσβαση σε κάθε κλήση του API του S3, ενώ το `aws s3` έχει μόνο τις συνηθισμένες εντολές για αρχεία. Το `sort_by` ταξινομεί τα tags κατά κλειδί. Το bucket έχει τρία tags, ακριβώς όσα έδειχνε το `tags_all`.

ΚΡΑΤΑ ΑΥΤΟ

Τα `default_tags` πάνε σε κάθε resource που φτιάχνει αυτή η ρύθμιση του provider `aws`. Το `tags_all` είναι η ένωση με τα `tags` του resource και σε σύγκρουση κλειδιών κερδίζει το resource.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Ο provider στέλνει τα tags ενός νέου bucket μαζί με την αίτηση δημιουργίας και το AWS τα γράφει αμέσως. Το Moto 5.2.3 τα αγνοεί σε αυτή την αίτηση και τα δέχεται μόνο σε bucket που υπάρχει ήδη, όπως το `notes`. Αν στο Moto φτιάξεις νέο bucket με tags, το επόμενο plan θα δείξει ότι λείπουν και ένα δεύτερο apply τα γράφει.

Δύο regions με aliases

Το κεφάλαιο 5 έδειξε τη σύνταξη του `alias` με τον `random`. Εδώ αποκτά νόημα. Ένα αντίγραφο ασφαλείας ανήκει σε άλλο region από τα δεδομένα, ώστε να αντέξει βλάβη ολόκληρου του region. Φτιάξε νέο φάκελο με το lock file του `first`.

TERMINAL

```
cd ..
mkdir regions
cd regions
cp ../first/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}
```

```
provider "aws" {
  region = "eu-central-1"
}

provider "aws" {
  alias  = "ireland"
  region = "eu-west-1"
}

data "aws_caller_identity" "current" {}

locals {
  prefix = "tfbook-${data.aws_caller_identity.current.account_id}"
}

resource "aws_s3_bucket" "data" {
  bucket = "${local.prefix}-data"
}

resource "aws_s3_bucket" "backup" {
  provider = aws.ireland
  bucket   = "${local.prefix}-backup"
}
```

Το πρώτο block `provider` είναι η προεπιλεγμένη ρύθμιση, στη Φρανκφούρτη. Το δεύτερο έχει `alias = "ireland"`. Το `aws_s3_bucket.data` δεν γράφει `provider`, οπότε φτιάχνεται στη Φρανκφούρτη. Το `aws_s3_bucket.backup` ζητά τη δεύτερη ρύθμιση με `provider = aws.ireland`, χωρίς εισαγωγικά. Το data source ρωτά την προεπιλεγμένη, αλλά ο λογαριασμός είναι ίδιος και στις δύο.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v6.67.0...
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

Οι δύο ρυθμίσεις μοιράζονται το ίδιο εκτελέσιμο, που εγκαταστάθηκε μία φορά.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.data: Creating...
aws_s3_bucket.backup: Creating...
aws_s3_bucket.data: Creation complete after 0s [id=tfbook-123456789012-data]
aws_s3_bucket.backup: Creation complete after 3s [id=tfbook-123456789012-backup]
…
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

TERMINAL

```
aws s3api get-bucket-location --bucket "tfbook-$ACCOUNT-data" --output text
aws s3api get-bucket-location --bucket "tfbook-$ACCOUNT-backup" --output text
```

ΕΞΟΔΟΣ

```
eu-central-1
eu-west-1
```

Το `get-bucket-location` ρωτά το AWS σε ποιο region ζει ένα bucket. Το `data` είναι στο `eu-central-1` και το `backup` στο `eu-west-1`. Ένα state και ένα apply έφτιαξαν resources σε δύο regions.

ΣΗΜΕΙΩΣΗ

Από την έκδοση 6 ο AWS provider δέχεται argument `region` στα περισσότερα resources. Ένα `region = "eu-west-1"` στο ίδιο το bucket το στέλνει στην Ιρλανδία χωρίς alias. Είναι δυνατότητα μόνο του AWS provider. Τα aliases είναι μηχανισμός του Terraform, δουλεύουν σε κάθε provider και είναι αυτά που ρωτά η εξέταση.

Aliases και modules

Το κεφάλαιο 15 έδειξε ότι ένα child module δεν έχει δικό του block `provider` και παίρνει τη ρύθμιση από το root. Φτιάξε ένα μικρό module για bucket καταγραφής.

modules/bucket/versions.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.67.0"
    }
  }
}
```

modules/bucket/main.tf

```
variable "name" {
  type        = string
  description = "Το όνομα του bucket"
}

resource "aws_s3_bucket" "this" {
  bucket = var.name
}

output "region" {
  value       = aws_s3_bucket.this.region
  description = "Το region όπου φτιάχτηκε το bucket"
}
```

Το module δεν ξέρει σε ποιο region θα βρεθεί. Το αποφασίζει ο καλών. Γράψε δύο κλήσεις στο `logs.tf` του root.

logs.tf

```
module "logs" {
  source = "./modules/bucket"
  name   = "${local.prefix}-logs"
}

module "logs_ireland" {
  source = "./modules/bucket"
  name   = "${local.prefix}-logs-ireland"

  providers = {
    aws = aws.ireland
  }
}

output "log_regions" {
  value = {
    logs         = module.logs.region
    logs_ireland = module.logs_ireland.region
  }
}
```

Η κλήση `logs` δεν γράφει `providers`, οπότε το child παίρνει την προεπιλεγμένη ρύθμιση του root. Η κλήση `logs_ireland` περνά με το `providers` τη ρύθμιση `aws.ireland`. Το κλειδί `aws` είναι το τοπικό όνομα μέσα στο child και η τιμή η ρύθμιση του root. Για το child η Ιρλανδία είναι απλώς ο `aws` του.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- logs in modules/bucket
- logs_ireland in modules/bucket
…
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v6.67.0
…
Terraform has been successfully initialized!
…
```

Το init βρήκε τις δύο κλήσεις. Τον provider τον είχε ήδη, γι' αυτό γράφει `Using previously-installed`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
module.logs_ireland.aws_s3_bucket.this: Creating...
module.logs.aws_s3_bucket.this: Creating...
module.logs_ireland.aws_s3_bucket.this: Creation complete after 0s [id=tfbook-123456789012-logs-ireland]
module.logs.aws_s3_bucket.this: Creation complete after 0s [id=tfbook-123456789012-logs]
…
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
…
log_regions = {
  "logs" = "eu-central-1"
  "logs_ireland" = "eu-west-1"
}
```

Το ίδιο module έφτιαξε ένα bucket σε κάθε region.

Ένα module μπορεί να χρειάζεται δύο ρυθμίσεις ταυτόχρονα, για παράδειγμα για να φτιάξει ένα bucket και το αντίγραφό του σε άλλο region. Τότε δηλώνει τα ονόματα που περιμένει με `configuration_aliases` στο `required_providers` και τα resources του γράφουν `provider = aws.primary` ή `provider = aws.secondary`.

HCL

```
terraform {
  required_providers {
    aws = {
      source                = "hashicorp/aws"
      version               = ">= 6.67.0"
      configuration_aliases = [aws.primary, aws.secondary]
    }
  }
}
```

Η κλήση δίνει τότε και τις δύο, για παράδειγμα `providers = { aws.primary = aws, aws.secondary = aws.ireland }`. Η άσκηση 17.7 χτίζει ένα τέτοιο module.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Ένα child module χωρίς `providers` κληρονομεί την προεπιλεγμένη ρύθμιση κάθε provider, το block χωρίς `alias`. Μια ρύθμιση με alias δεν κληρονομείται ποτέ αυτόματα. Περνά μόνο ρητά, με το `providers` στο block `module`. Το κλειδί του map είναι το όνομα μέσα στο child και η τιμή η ρύθμιση του parent.

Ένα workflow για πολλούς providers

Το κεφάλαιο 5 μίλησε για multi cloud και hybrid cloud. Η ουσία τους είναι ότι μία γλώσσα, ένα workflow και ένα state καλύπτουν όλες τις πλατφόρμες. Εδώ το βλέπεις με τρεις providers. Ο `random` δίνει μοναδικό κομμάτι για το όνομα του bucket, ο δεύτερος συνηθισμένος τρόπος μετά τον αριθμό του λογαριασμού. Ο `aws` φτιάχνει το bucket. Ο `local` γράφει τα στοιχεία του σε αρχείο, όπως θα τα έγραφε ένας provider για το σύστημα του datacenter σου.

TERMINAL

```
cd ..
mkdir mixed
cd mixed
cp ../first/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "random_pet" "suffix" {
  length = 2
}

resource "aws_s3_bucket" "site" {
  bucket = "tfbook-${random_pet.suffix.id}"
}

resource "local_file" "info" {
  filename = "bucket.txt"
  content  = <<-EOT
    name   = ${aws_s3_bucket.site.bucket}
    region = ${aws_s3_bucket.site.region}
    arn    = ${aws_s3_bucket.site.arn}
```

```
  EOT
}

output "bucket" {
  value = aws_s3_bucket.site.bucket
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Reusing previous version of hashicorp/aws from the dependency lock file
- Finding hashicorp/random versions matching "3.9.1"...
- Finding hashicorp/local versions matching "2.9.1"...
- Installing hashicorp/aws v6.67.0...
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)
- Installing hashicorp/local v2.9.1...
- Installed hashicorp/local v2.9.1 (signed by HashiCorp)
…
```

Την έκδοση του `aws` το init την πήρε από το lock file που αντέγραψες. Τον `random` και τον `local` τους βρήκε στο Registry και τους πρόσθεσε στο lock file.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.suffix: Creating...
random_pet.suffix: Creation complete after 0s [id=tender-flounder]
aws_s3_bucket.site: Creating...
aws_s3_bucket.site: Creation complete after 0s [id=tfbook-tender-flounder]
local_file.info: Creating...
local_file.info: Creation complete after 0s [id=bc403f3e6f2a9db2602d6f3b0fa56fef1bd21428]
…
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
…
```

Η σειρά της εξόδου είναι η σειρά του graph. Πρώτα το `random_pet`, γιατί το όνομα του bucket το χρειάζεται. Μετά το bucket. Τελευταίο το `local_file`, γιατί το περιεχόμενό του διαβάζει το `arn` του bucket, που υπάρχει μόνο μετά τη δημιουργία. Ο Core δεν ξεχωρίζει providers όταν χτίζει το graph. Βλέπει μόνο references.

TERMINAL

```
cat bucket.txt
terraform state list
```

ΕΞΟΔΟΣ

```
name   = tfbook-tender-flounder
region = eu-central-1
arn    = arn:aws:s3:::tfbook-tender-flounder
aws_s3_bucket.site
local_file.info
random_pet.suffix
```

Το όνομα του bucket έχει δύο τυχαίες λέξεις, που στη δική σου εκτέλεση θα είναι άλλες. Το state κρατά μαζί τα resources και των τριών providers. Με τον provider του Azure ή του VMware vSphere στη θέση του `local` το workflow θα ήταν ακριβώς το ίδιο.

ΚΡΑΤΑ ΑΥΤΟ

Multi cloud με Terraform σημαίνει πολλούς providers σε μία configuration, ένα graph, ένα plan και ένα state. Δεν σημαίνει ότι ένα resource του AWS μεταφράζεται σε resource άλλου cloud. Κάθε provider έχει τους δικούς του τύπους resources.

Κόστος και καθάρισμα

Στο Moto τίποτα δεν κοστίζει. Στο AWS κάθε resource που ξεχνάς χρεώνεται μέχρι να το σβήσεις.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Ένα άδειο bucket δεν χρεώνεται και τα λίγα bytes αυτού του κεφαλαίου κοστίζουν ελάχιστα. Οι servers των επόμενων κεφαλαίων όμως χρεώνονται για κάθε ώρα λειτουργίας. Κάνε `terraform destroy` σε κάθε φάκελο στο τέλος του κεφαλαίου και επιβεβαίωσε με το `aws` CLI ότι δεν έμεινε τίποτα. Το budget της αρχής πιάνει ό,τι ξεχάσεις.

Σβήσε και τους τρεις φακέλους.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 3 destroyed.
```

TERMINAL

```
cd ../regions
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 4 destroyed.
```

TERMINAL

```
cd ../first
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_object.hello: Destruction complete after 0s
…
aws_s3_bucket.notes: Destruction complete after 0s
…
Destroy complete! Resources: 2 destroyed.
```

TERMINAL

```
aws s3 ls
```

Στο `first` το destroy έσβησε πρώτα το object και μετά το bucket, γιατί ένα bucket σβήνεται μόνο όταν είναι άδειο. Το `aws s3 ls` δεν τυπώνει τίποτα, άρα δεν έμεινε κανένα bucket.

ΠΑΓΙΔΑ

Το Terraform σβήνει μόνο τα objects που έχει στο state. Αν κάποιος ανέβασε αρχείο στο bucket με άλλο τρόπο, το destroy σταματά με `BucketNotEmpty`. Το argument `force_destroy = true` λέει στον provider να αδειάσει πρώτα το bucket. Πρέπει όμως να γραφτεί στο state με ένα apply πριν από το destroy. Η άσκηση 17.5 το δείχνει.

Ασκήσεις

Οι ασκήσεις ξεκινούν από αντίγραφο ενός φακέλου του κεφαλαίου, ώστε να υπάρχει το lock file. Όποια κάνει apply τελειώνει με `terraform destroy -auto-approve`.

ΑΣΚΗΣΗ 17.1 Το name που έγινε region

Ξεκίνα από αντίγραφο του φακέλου `first`. Πρόσθεσε στο τέλος του `identity.tf` ένα output `region_name` με τιμή `data.aws_region.current.name`. Τρέξε `terraform init` και `terraform plan`. Τι λέει το Terraform για το νέο output;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  + region_name = "eu-central-1"
…
│ Warning: Deprecated value used
…
│   on identity.tf line 18, in output "region_name":
…
│   The deprecation originates from data.aws_region.current.name
…
│ name is deprecated. Use region instead.
…
```

ΑΣΚΗΣΗ 17.2 Ποιο region κερδίζει

Ξεκίνα από αντίγραφο του φακέλου `first`. Σβήσε το argument `region` από το block `provider` του `main.tf` και κράτα τα `default_tags`. Τρέξε init και μετά `AWS_REGION=eu-west-1 AWS_DEFAULT_REGION=eu-north-1 terraform plan`. Σε ποιο region θα δούλευε ο provider;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  + region     = "eu-west-1"
…
```

ΑΣΚΗΣΗ 17.3 Τα στοιχεία μέσα στο bucket

Ξεκίνα από αντίγραφο του φακέλου `mixed`. Πρόσθεσε ένα `aws_s3_object` με όνομα `info` που ανεβάζει στο bucket το περιεχόμενο του `local_file.info` με key `bucket.txt`. Κάνε apply και δείξε τα resources του state.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_s3_bucket.site
aws_s3_object.info
local_file.info
random_pet.suffix
```

ΑΣΚΗΣΗ 17.4 Tags μόνο στον AWS

Ξεκίνα από αντίγραφο του φακέλου `mixed` και κάνε apply. Μετά πρόσθεσε στο block `provider "aws"` τα `default_tags` με `Project = "tfbook"` και `Chapter = "17"`. Τρέξε `terraform plan`. Ποια resources θα αλλάξουν;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.site will be updated in-place
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 17.5 Το bucket που δεν άδειαζε

Ξεκίνα από αντίγραφο του φακέλου `mixed` και κάνε apply. Ανέβασε ένα αρχείο στο bucket με το `aws` CLI, για παράδειγμα με `aws s3 cp --no-progress main.tf "s3://$(terraform output -raw bucket)/main.tf"`. Δοκίμασε `terraform destroy`. Μετά κάνε ό,τι χρειάζεται ώστε το destroy να πετύχει χωρίς να σβήσεις το αρχείο με το χέρι.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Destroy complete! Resources: 3 destroyed.
```

ΑΣΚΗΣΗ 17.6 Τρίτο region

Ξεκίνα από αντίγραφο του φακέλου `regions`. Πρόσθεσε σε νέο αρχείο `stockholm.tf` μια ρύθμιση του `aws` με alias `stockholm` στο `eu-north-1` και ένα bucket `${local.prefix}-stockholm` εκεί. Κάνε apply και έλεγξε το region του νέου bucket με `aws s3api get-bucket-location`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
eu-north-1
```

ΑΣΚΗΣΗ 17.7 Ένα module, δύο regions

Ξεκίνα από αντίγραφο του φακέλου `regions`. Γράψε module στο `modules/pair` που δέχεται μεταβλητή `name` και φτιάχνει δύο buckets, το `${var.name}-primary` με τη ρύθμιση `aws.primary` και το `${var.name}-secondary` με τη ρύθμιση `aws.secondary`. Δώσε output `regions` με map από το όνομα κάθε bucket στο region του. Κάλεσέ το από νέο αρχείο `pair.tf` με όνομα `${local.prefix}-pair`, primary τη Φρανκφούρτη και secondary την Ιρλανδία. Βγάλε το map στο root με output `pair_regions`, κάνε apply και δείξε το με `terraform output`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
{
  "tfbook-123456789012-pair-primary" = "eu-central-1"
  "tfbook-123456789012-pair-secondary" = "eu-west-1"
}
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 17.1

Μια ομάδα τρέχει Terraform από laptops και από ένα σύστημα CI. Ποιος τρόπος να δοθούν credentials στον AWS provider δεν συνιστάται;

a. Τα `access_key` και `secret_key` γραμμένα στο block `provider`.
b. Ένα profile του SSO που διαλέγεται με τη μεταβλητή `AWS_PROFILE`.
c. Οι μεταβλητές `AWS_ACCESS_KEY_ID` και `AWS_SECRET_ACCESS_KEY` από τα secrets του CI.
d. Ο IAM role του server του EC2 όπου τρέχει το CI.

ΕΡΩΤΗΣΗ 17.2

Ο provider έχει `default_tags` με `Environment = "dev"` και `Team = "data"`. Ένα `aws_s3_bucket` έχει `tags = { Environment = "prod", Name = "logs" }`. Ποια tags θα έχει το bucket στο AWS;

a. `Environment = "dev"`, `Team = "data"` και `Name = "logs"`
b. `Environment = "prod"` και `Name = "logs"`
c. `Environment = "prod"`, `Team = "data"` και `Name = "logs"`
d. Το plan αποτυγχάνει επειδή το `Environment` ορίζεται δύο φορές.

ΕΡΩΤΗΣΗ 17.3

Το root module έχει `provider "aws"` με `region = "eu-central-1"` και δεύτερο `provider "aws"` με `alias = "west"` και `region = "eu-west-1"`. Καλεί ένα module με `providers = { aws = aws.west }`. Το module έχει ένα `aws_s3_bucket` χωρίς argument `provider`. Σε ποιο region φτιάχνεται το bucket;

a. Στο `eu-central-1`, γιατί τα modules παίρνουν πάντα την προεπιλεγμένη ρύθμιση.
b. Στο `eu-west-1`, γιατί για το module η ρύθμιση `aws.west` είναι ο `aws` του.
c. Σε κανένα, γιατί ρύθμιση με alias δεν περνά σε module.
d. Και στα δύο regions, ένα bucket σε καθένα.

ΕΡΩΤΗΣΗ 17.4

Μια εταιρεία έχει servers σε δικό της datacenter με VMware vSphere, εφαρμογές στο AWS και DNS σε άλλον πάροχο. Ποια δύο ισχύουν για τη διαχείρισή τους με Terraform; Επίλεξε δύο.

a. Μία configuration μπορεί να έχει resources και των τριών providers, με references ανάμεσά τους.
b. Το Terraform μεταφράζει αυτόματα ένα `aws_instance` σε virtual machine του vSphere.
c. Κάθε provider χρειάζεται υποχρεωτικά δικό του χωριστό state.
d. Το `init`, το `plan` και το `apply` δουλεύουν με τον ίδιο τρόπο για όλους τους providers.

ΕΡΩΤΗΣΗ 17.5

Σωστό ή λάθος; Το block `provider "aws"` γράφει `region = "eu-central-1"`. Αν τρέξεις `AWS_REGION=us-east-1 terraform apply`, τα resources φτιάχνονται στο `us-east-1`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 17.6

Το Terraform τρέχει σε server του EC2 που έχει IAM role. Δεν υπάρχουν credentials στο block `provider`, σε μεταβλητές περιβάλλοντος ή στα αρχεία του `~/.aws`. Τι κάνει ο AWS provider;

a. Σταματά αμέσως με `No valid credential sources found`.
b. Ζητά από τον χρήστη να γράψει access key στο τερματικό.
c. Παίρνει προσωρινά credentials του role από την υπηρεσία metadata του server.
d. Κάνει τις κλήσεις ανώνυμα, χωρίς υπογραφή.



ΚΕΦΑΛΑΙΟ 18

Δίκτυο και ένας web server

Στο κεφάλαιο 17 έφτιαξες ένα bucket. Εδώ στήνεις κάτι που μοιάζει με πραγματική υποδομή. Ένα δίκτυο με δική του έξοδο στο internet, ένα firewall και έναν server που σηκώνει μια ιστοσελίδα μόλις ξεκινήσει. Στην πορεία βλέπεις τις αναφορές και τις εξαρτήσεις του κεφαλαίου 6 σε ένα stack με εννέα resources και κλείνεις το δίκτυο σε ένα module.

Το περιβάλλον

Το περιβάλλον είναι αυτό του κεφαλαίου 17. Το Moto τρέχει σε Docker στη θύρα 5050 και το τερματικό σου έχει τις μεταβλητές που στέλνουν τον AWS provider και το `aws` CLI εκεί. Αν άνοιξες νέο τερματικό, όρισέ τις ξανά.

TERMINAL

```
export AWS_ENDPOINT_URL=http://localhost:5050
export AWS_ACCESS_KEY_ID=test
export AWS_SECRET_ACCESS_KEY=test
export AWS_REGION=eu-central-1
```

Με sandbox λογαριασμό δεν ορίζεις αυτές τις μεταβλητές. Κάνεις `aws sso login --profile tfbook` και `export AWS_PROFILE=tfbook`, όπως στο κεφάλαιο 17.

Έλεγξε ότι το CLI μιλά με το Moto.

TERMINAL

```
echo $AWS_ENDPOINT_URL
aws sts get-caller-identity --query Account --output text
```

ΕΞΟΔΟΣ

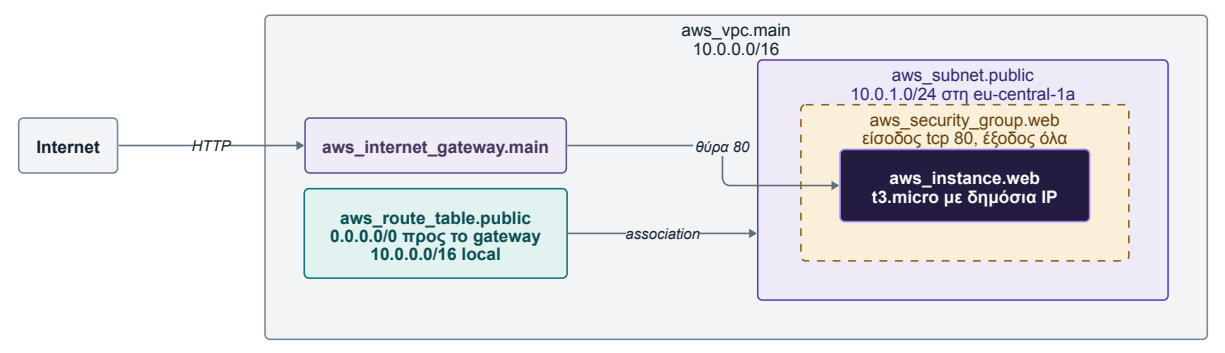
```
http://localhost:5050
123456789012
```

Ο `123456789012` είναι ο λογαριασμός του Moto. Σε sandbox λογαριασμό βλέπεις τον δικό σου αριθμό. Ο κώδικας του κεφαλαίου είναι ίδιος και στις δύο περιπτώσεις.

Τα κομμάτια του δικτύου

Ένας server χρειάζεται δίκτυο, δρόμο προς το internet και κανόνες για το ποιος του μιλά. Στην κονσόλα πολλά από αυτά τα φτιάχνει σιωπηλά ο οδηγός. Στο Terraform τα γράφεις ένα ένα.

- **Region και availability zones.** Το region `eu-central-1` είναι η Φρανκφούρτη. Έχει τρεις availability zones, όπως η `eu-central-1a`. Κάθε zone είναι ένα ή περισσότερα κέντρα δεδομένων με δικό τους ρεύμα και δίκτυο, χωριστά από τις άλλες zones.
- **VPC** (Virtual Private Cloud). Το δικό σου απομονωμένο δίκτυο μέσα σε ένα region. Το CIDR block λέει ποιες διευθύνσεις του ανήκουν. Το `10.0.0.0/16` κρατά σταθερά τα πρώτα 16 bits, άρα δίνει 65.536 διευθύνσεις, από `10.0.0.0` έως `10.0.255.255`. Είναι ιδιωτικές διευθύνσεις που δεν φαίνονται από το internet.
- **Subnet.** Ένα κομμάτι του VPC σε μία μόνο availability zone. Το `10.0.1.0/24` έχει 256 διευθύνσεις. Το AWS κρατά πέντε για δική του χρήση.
- **Internet gateway.** Η πόρτα του VPC προς το internet. Ένα ανά VPC.
- **Route table.** Κανόνες της μορφής «για αυτόν τον προορισμό, στείλε εκεί». Κάθε route table έχει αυτόματα τον κανόνα `local` για τις διευθύνσεις του VPC. Εμείς προσθέτουμε τον `0.0.0.0/0`, δηλαδή «οτιδήποτε άλλο», προς το internet gateway. Ένα **association** εφαρμόζει το route table σε ένα subnet. Ένα subnet που στέλνει το `0.0.0.0/0` σε internet gateway λέγεται **δημόσιο** (public).
- **Security group.** Firewall γύρω από τον server. Χωρίς κανόνες απορρίπτει κάθε εισερχόμενη σύνδεση. Είναι stateful. Όταν επιτρέπεις μια σύνδεση, η απάντηση γυρίζει χωρίς δικό της κανόνα.



Σχήμα 18.1 · Η τοπολογία του κεφαλαίου. Το route table στέλνει ό,τι δεν είναι τοπικό στο internet gateway και το association το εφαρμόζει στο subnet. Ο server παίρνει δημόσια IP από το subnet και το security group αφήνει να περάσει μόνο η θύρα 80.

Ο φάκελος και ο provider

TERMINAL

```
mkdir web
cd web
```

Το block `terraform` και το block `provider` μπαίνουν στο `versions.tf`. Τα resources μοιράζονται σε αρχεία ανά ρόλο, όπως στο κεφάλαιο 8.

versions.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Finding hashicorp/aws versions matching "6.67.0"...
- Installing hashicorp/aws v6.67.0...
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

Ο φάκελος δεν έχει ακόμη lock file, οπότε το init κατεβάζει τον provider ακόμη κι αν έχεις το plugin cache του κεφαλαίου 17. Οι επόμενοι φάκελοι του κεφαλαίου θα αντιγράψουν το lock file του `web`.

Το VPC και το subnet

Το `network.tf` ξεκινά με το VPC και το subnet.

network.tf

```
data "aws_availability_zones" "available" {
  state = "available"
}

resource "aws_vpc" "main" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true
```

```
  tags = {
    Name = "tfbook-web"
  }
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = data.aws_availability_zones.available.names[0]
  map_public_ip_on_launch = true

  tags = {
    Name = "tfbook-web-public"
  }
}
```

Διάβασέ το block προς block.

- Το data source `aws_availability_zones` ρωτά ποιες availability zones έχει το region του provider. Με `state = "available"` κρατά μόνο όσες λειτουργούν κανονικά. Το attribute `names` είναι λίστα με τα ονόματά τους.
- Το `aws_vpc.main` παίρνει το CIDR block του VPC. Το `enable_dns_hostnames = true` δίνει σε κάθε server με δημόσια IP και ένα δημόσιο όνομα DNS. Το tag `Name` είναι το όνομα που δείχνει η κονσόλα.
- Το `aws_subnet.public` αναφέρει το `aws_vpc.main.id`. Το ID του VPC δεν υπάρχει πριν φτιαχτεί το VPC, οπότε η αναφορά είναι και implicit dependency. Η zone έρχεται από το data source με index `[0]`, το πρώτο στοιχείο της λίστας (κεφάλαιο 9). Το `map_public_ip_on_launch = true` λέει στο AWS να δίνει δημόσια IP σε κάθε server που ξεκινά σε αυτό το subnet.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Η αναφορά σε data source ξεκινά πάντα με `data.`, όπως το `data.aws_availability_zones.available.names`. Η αναφορά σε resource δεν έχει πρόθεμα, όπως το `aws_vpc.main.id`. Η εξέταση δείχνει τέτοιες εκφράσεις και ρωτά αν πρόκειται για κάτι που το Terraform φτιάχνει ή για κάτι που μόνο διαβάζει. Το block `resource` φτιάχνει και διαχειρίζεται, το block `data` διαβάζει κάτι που υπάρχει ήδη (κεφάλαιο 12).

Πρόσθεσε στο ίδιο αρχείο την έξοδο προς το internet.

network.tf

```
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
}

resource "aws_route_table_association" "public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```



Το internet gateway χρειάζεται μόνο το VPC του. Το route table έχει ένα nested block `route` με τον κανόνα. Το `cidr_block` είναι ο προορισμός και το `gateway_id` ο δρόμος. Τον κανόνα `local` τον βάζει το AWS μόνο του. Το association απλώς ενώνει subnet και route table. Είναι χωριστό resource, γιατί η σύνδεση φτιάχνεται και σβήνεται ανεξάρτητα από τα δύο άκρα της.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Ένα subnet είναι δημόσιο μόνο αν το route table του στέλνει το `0.0.0.0/0` σε internet gateway. Χωρίς association το subnet χρησιμοποιεί το main route table του VPC, που έχει μόνο τον κανόνα `local`. Ο server τότε δεν βγαίνει στο internet, όσες δημόσιες IP κι αν έχει.

Το security group

Το firewall γράφεται σε τρία resources, ένα για την ομάδα και ένα για κάθε κανόνα.

security.tf

```
resource "aws_security_group" "web" {
  name        = "tfbook-web"
  description = "HTTP from anywhere"
  vpc_id      = aws_vpc.main.id
}

resource "aws_vpc_security_group_ingress_rule" "http" {
  security_group_id = aws_security_group.web.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "tcp"
  from_port         = 80
  to_port           = 80
}

resource "aws_vpc_security_group_egress_rule" "all" {
  security_group_id = aws_security_group.web.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "-1"
}
```

- Το `aws_security_group.web` ζει στο VPC μας. Το `name` πρέπει να είναι μοναδικό μέσα στο VPC.
- Ο κανόνας `http` επιτρέπει εισερχόμενες συνδέσεις TCP στη θύρα 80 από κάθε διεύθυνση IPv4. Τα `from_port` και `to_port` ορίζουν εύρος θυρών. Εδώ είναι μία θύρα, οπότε είναι ίσα.

- Ο κανόνας `all` επιτρέπει κάθε εξερχόμενη σύνδεση. Το `ip_protocol = "-1"` σημαίνει όλα τα πρωτόκολλα και όλες τις θύρες, οπότε θύρες δεν γράφεις.

Το AWS βάζει σε κάθε νέο security group έναν κανόνα εξόδου που επιτρέπει τα πάντα. Το resource `aws_security_group` του Terraform τον σβήνει όταν φτιάχνει την ομάδα, ώστε η έξοδος να περιγράφεται κι αυτή στον κώδικα. Γι' αυτό γράφεις τον κανόνα `all`. Χωρίς αυτόν ο server δεν θα μπορούσε να κατεβάσει πακέτα.

> **ΠΑΓΙΔΑ**
> Σε παλιότερο κώδικα θα δεις κανόνες μέσα στο `aws_security_group`, σε nested blocks `ingress` και `egress`, ή το resource `aws_security_group_rule`. Μην ανακατεύεις τους τρόπους στην ίδια ομάδα. Το καθένα σβήνει ό,τι έγραψε το άλλο και κάθε plan βρίσκει διαφορές. Τα χωριστά resources κανόνων είναι ο τρόπος που προτείνει σήμερα ο provider. Κάθε κανόνας έχει δική του διεύθυνση, οπότε ένας νέος κανόνας δεν αγγίζει τους άλλους.

Η εικόνα του server

Ένας server ξεκινά από ένα **AMI** (Amazon Machine Image), ένα έτοιμο image δίσκου με λειτουργικό σύστημα. Το ID ενός AMI είναι διαφορετικό σε κάθε region και αλλάζει με κάθε νέα έκδοση. Αντί να το γράψεις με το χέρι, το βρίσκεις με το data source `aws_ami`.

server.tf

```
data "aws_ami" "al2023" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["al2023-ami-2023.*-kernel-6.1-x86_64"]
  }
}
```

- Το `owners` περιορίζει την αναζήτηση στα images που δημοσιεύει η ίδια η Amazon. Το `"amazon"` είναι ψευδώνυμο που δέχεται το API στη θέση του αριθμού λογαριασμού.
- Το block `filter` ψάχνει στο όνομα. Το `*` ταιριάζει με οτιδήποτε. Τα standard images του Amazon Linux 2023 λέγονται, για παράδειγμα, `al2023-ami-2023.12.20260727.0-kernel-6.1-x86_64`. Το μοτίβο κρατά την έκδοση με kernel 6.1 για επεξεργαστές `x86_64`, όποια ημερομηνία κι αν έχει.
- Το `most_recent = true` διαλέγει το νεότερο αν ταιριάζουν πολλά. Χωρίς αυτό, πολλά αποτελέσματα είναι σφάλμα.

Το μοτίβο δουλεύει στο Moto και στο AWS. Το Moto κρατά μια λίστα με πραγματικά images, ανάμεσά τους και αυτό.

> **ΠΑΓΙΔΑ**
> Χωρίς `owners` η αναζήτηση ψάχνει και σε δημόσια images άλλων λογαριασμών. Οποιοσδήποτε μπορεί να δημοσιεύσει image με όνομα που μοιάζει με της Amazon. Γράφε πάντα `owners`. Επίσης, η αρχιτεκτονική του image πρέπει να ταιριάζει με το instance type. Τα `t3` έχουν επεξεργαστές `x86_64`, οπότε ένα image `arm64` δεν ξεκινά σε αυτά.

Το user_data από template

Το **user_data** είναι κείμενο που δίνεις στον server όταν τον φτιάχνεις. Στο Amazon Linux το διαβάζει το cloud-init. Αν ξεκινά με `#!`, το εκτελεί ως script με δικαιώματα root, μία φορά, στην πρώτη εκκίνηση. Το script γράφεται σε δικό του αρχείο template, όπως στο κεφάλαιο 10.

user_data.sh.tftpl

```
#!/bin/bash
dnf install -y httpd
cat > /var/www/html/index.html <<'HTML'
<h1>${title}</h1>
HTML
systemctl enable --now httpd
```

Το `dnf` εγκαθιστά τον Apache httpd. Το `cat` γράφει τη σελίδα στον φάκελο που σερβίρει ο httpd. Το `systemctl enable --now` ξεκινά τον httpd τώρα και σε κάθε επόμενη εκκίνηση. Το μόνο κομμάτι του Terraform είναι το `${title}`. Η `templatefile` το αντικαθιστά πριν φτάσει οτιδήποτε στο AWS, οπότε ο server παίρνει έτοιμο script.

Το template θεωρεί κάθε `${...}` δική του παρεμβολή. Για μεταβλητή του bash γράψε `$HOME` χωρίς άγκιστρα ή `$${HOME}`, που το template αφήνει ως `${HOME}`.

Ο server

Πρόσθεσε τον server στο `server.tf`, κάτω από το data source.

server.tf

```
resource "aws_instance" "web" {
  ami                    = data.aws_ami.al2023.id
  instance_type          = "t3.micro"
  subnet_id              = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.web.id]

  user_data = templatefile("${path.module}/user_data.sh.tftpl", {
    title = "Γεια σου από το Terraform"
  })
  user_data_replace_on_change = true

  tags = {
    Name = "tfbook-web"
  }

  depends_on = [aws_route_table_association.public]
}
```

Κάθε γραμμή είναι μια αναφορά ή μια ρύθμιση.

- Το `ami` παίρνει το ID από το data source.
- Το `instance_type` είναι το μέγεθος. Το `t3.micro` έχει δύο vCPU και 1 GiB μνήμη.
- Το `subnet_id` βάζει τον server στο δημόσιο subnet και άρα στη zone του.
- Το `vpc_security_group_ids` είναι λίστα, γιατί ένας server μπορεί να έχει πολλά security groups.
- Το `user_data` είναι το αποτέλεσμα της `templatefile`. Το `user_data_replace_on_change` εξηγείται παρακάτω.

Το `depends_on` είναι η περίπτωση του κεφαλαίου 6, μια εξάρτηση που δεν περνά από αναφορά. Το script χρειάζεται internet από τα πρώτα δευτερόλεπτα, για το `dnf install`. Το internet φτάνει στο subnet μόνο

όταν υπάρξει το association. Ο server όμως αναφέρει το subnet, όχι το association. Χωρίς το `depends_on` το Terraform θα μπορούσε να ξεκινήσει τον server ταυτόχρονα με το association. Συνήθως η εκκίνηση αργεί αρκετά ώστε να μη φανεί πρόβλημα, αλλά το «συνήθως» δεν είναι εγγύηση.

Τα outputs δίνουν ό,τι θα χρειαστείς μετά το apply.

outputs.tf

```
output "instance_id" {
  value       = aws_instance.web.id
  description = "Το ID του server"
}

output "public_ip" {
  value       = aws_instance.web.public_ip
  description = "Η δημόσια IPv4 του server"
}

output "url" {
  value       = "http://${aws_instance.web.public_ip}"
  description = "Η διεύθυνση της σελίδας"
}
```

Το plan

Αποθήκευσε το plan σε αρχείο, όπως στο κεφάλαιο 3. Θα το διαβάσεις από δύο μεριές και μετά θα εφαρμόσεις ακριβώς αυτό. Η έξοδος κρατά τις γραμμές με τα data sources, τις επικεφαλίδες των resources και την περίληψη.

TERMINAL

```
terraform plan -out=web.tfplan
```

ΕΞΟΔΟΣ

```
…
data.aws_availability_zones.available: Read complete after 1s [id=eu-central-1]
data.aws_ami.al2023: Read complete after 1s [id=ami-0352a6b853b4367b3]
…
  # aws_instance.web will be created
…
  # aws_internet_gateway.main will be created
…
  # aws_route_table.public will be created
…
  # aws_route_table_association.public will be created
…
  # aws_security_group.web will be created
…
  # aws_subnet.public will be created
…
  # aws_vpc.main will be created
…
  # aws_vpc_security_group_egress_rule.all will be created
…
  # aws_vpc_security_group_ingress_rule.http will be created
…
Plan: 9 to add, 0 to change, 0 to destroy.
…
```

Τα δύο data sources διαβάστηκαν μέσα στο plan, γιατί τα arguments τους είναι γνωστά από πριν. Στις αγκύλες του `aws_ami` είναι το ID του image που βρέθηκε. Το `aws_availability_zones` έχει για `id` το όνομα του region. Η περίληψη λέει εννέα resources. Τα data sources δεν μετρούν, γιατί δεν φτιάχνουν τίποτα.

Το `terraform show` δείχνει το αποθηκευμένο plan. Εδώ κρατάμε μερικά attributes του server και του subnet.

TERMINAL

```
terraform show web.tfplan
```

ΕΞΟΔΟΣ

```
…
  # aws_instance.web will be created
…
      + ami                                  = "ami-0352a6b853b4367b3"
…
      + availability_zone                    = (known after apply)
…
      + public_ip                            = (known after apply)
…
      + subnet_id                            = (known after apply)
…
            #!/bin/bash
            dnf install -y httpd
            cat > /var/www/html/index.html <<'HTML'
            <h1>Γεια σου από το Terraform</h1>
            HTML
            systemctl enable --now httpd
…
  # aws_subnet.public will be created
…
      + availability_zone                              = "eu-central-1a"
…
```

Το `ami` έχει ήδη την τιμή του data source. Το ίδιο και το `availability_zone` του subnet, που ήρθε από τη λίστα των zones. Το `subnet_id` του server γράφει `(known after apply)`, γιατί το subnet δεν υπάρχει ακόμη. Το ίδιο και η `public_ip` και η zone του server, που θα τις δώσει το AWS. Το `user_data` φαίνεται ολόκληρο, με τον τίτλο ήδη στη θέση του `${title}`.

ΠΑΓΙΔΑ

Στον AWS provider 6 το `user_data` γράφεται στο plan και στο state όπως είναι. Το διαβάζει επίσης όποιος βλέπει τα attributes του server και κάθε πρόγραμμα μέσα σε αυτόν. Μη βάζεις κωδικούς στο user_data. Το κεφάλαιο 24 δείχνει πού μπαίνουν τα secrets.

Το apply και η σειρά δημιουργίας

TERMINAL

```
terraform apply web.tfplan
```



ΕΞΟΔΟΣ

```
aws_vpc.main: Creating...
aws_vpc.main: Creation complete after 0s [id=vpc-e621ae7c2bd43257c]
aws_internet_gateway.main: Creating...
aws_security_group.web: Creating...
aws_subnet.public: Creating...
aws_internet_gateway.main: Creation complete after 1s [id=igw-03bc255f512925e2a]
aws_route_table.public: Creating...
aws_security_group.web: Creation complete after 1s [id=sg-3fb0c6dcd2595d97c]
aws_vpc_security_group_ingress_rule.http: Creating...
aws_vpc_security_group_egress_rule.all: Creating...
aws_vpc_security_group_ingress_rule.http: Creation complete after 0s [id=sgr-e531dbe942a4452a2]
aws_vpc_security_group_egress_rule.all: Creation complete after 0s [id=sgr-c05f8b38e22a3f99f]
aws_route_table.public: Creation complete after 0s [id=rtb-8d8bd6c8476f6857d]
…
aws_subnet.public: Creation complete after 11s [id=subnet-1a6b4cc309482d72a]
aws_route_table_association.public: Creating...
aws_route_table_association.public: Creation complete after 0s [id=rtbassoc-da72417445e216324]
aws_instance.web: Creating...
…
aws_instance.web: Creation complete after 10s [id=i-f6260f341e77e8c90]
…
Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
…
```

Η έξοδος του apply είναι το graph του κεφαλαίου 6 σε εκτέλεση.

1. Πρώτο ξεκινά το `aws_vpc.main`, που δεν χρειάζεται κανένα resource.
2. Μόλις τελειώσει, ξεκινούν μαζί το internet gateway, το subnet και το security group. Και τα τρία περιμένουν μόνο το VPC.
3. Το route table ξεκινά μετά το internet gateway. Οι δύο κανόνες ξεκινούν μαζί μόλις υπάρξει το security group.
4. Το association περιμένει το subnet και το route table.
5. Ο server ξεκινά τελευταίος, αφού τελειώσει το association.

Η σειρά ανάμεσα σε resources του ίδιου βήματος αλλάζει από εκτέλεση σε εκτέλεση. Η σειρά ανάμεσα στα βήματα όχι. Μερικά resources κάνουν λίγα δευτερόλεπτα ακόμη και στο Moto, γιατί ο provider ρωτά ανά διαστήματα αν τελείωσαν. Ο server, για παράδειγμα, θεωρείται έτοιμος όταν φτάσει στην κατάσταση `running`.

TERMINAL

```
terraform output
```

ΕΞΟΔΟΣ

```
instance_id = "i-f6260f341e77e8c90"
public_ip = "54.214.247.179"
url = "http://54.214.247.179"
```

Το Moto έδωσε στον server δημόσια IP, επειδή το subnet έχει `map_public_ip_on_launch = true`. Η δική σου θα είναι άλλη.

Έλεγχος με το aws CLI

Το `aws` CLI δείχνει τον server από τη μεριά του AWS. Το `-raw` του κεφαλαίου 8 δίνει το ID χωρίς εισαγωγικά.

TERMINAL

```
ID=$(terraform output -raw instance_id)
aws ec2 describe-instances --instance-ids "$ID" \
  --query 'Reservations[0].Instances[0].[State.Name,InstanceType,PrivateIpAddress]' \
  --output text
```

ΕΞΟΔΟΣ

```
running t3.micro    10.0.1.4
```

Ο server είναι `running`, έχει το μέγεθος που ζήτησες και ιδιωτική IP από το `10.0.1.0/24`. Το AWS κρατά και το user_data, σε base64.

TERMINAL

```
aws ec2 describe-instance-attribute --instance-id "$ID" --attribute userData \
  --query UserData.Value --output text | base64 --decode
```

ΕΞΟΔΟΣ

```
#!/bin/bash
dnf install -y httpd
cat > /var/www/html/index.html <<'HTML'
<h1>Γεια σου από το Terraform</h1>
HTML
systemctl enable --now httpd
```

Είναι το script του template, με τον τίτλο στη θέση του.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Στο Moto ο server δεν εκτελεί τίποτα. Η κατάσταση `running` και η δημόσια IP είναι εγγραφές του εξομοιωτή. Το script δεν τρέχει και η IP δεν απαντά. Στο πραγματικό AWS το cloud-init τρέχει το script στην πρώτη εκκίνηση. Περίμενε ένα με δύο λεπτά μετά το apply και ζήτα τη σελίδα.

TERMINAL

```
curl "$(terraform output -raw url)"
```

Η απάντηση είναι το HTML του template. Αν το `curl` περιμένει χωρίς απάντηση, έλεγξε τον κανόνα της θύρας 80, το route και το association. Αν απαντά `Connection refused`, ο server φαίνεται αλλά ο httpd δεν ξεκίνησε. Τα μηνύματα του script είναι στο `/var/log/cloud-init-output.log` του server.

Το graph του stack

Το `terraform graph` του κεφαλαίου 6 δείχνει τις εξαρτήσεις. Κρατάμε μόνο τα βέλη.

TERMINAL

```
terraform graph
```

ΕΞΟΔΟΣ

```
…
  "aws_instance.web" -> "data.aws_ami.al2023";
  "aws_instance.web" -> "aws_route_table_association.public";
  "aws_instance.web" -> "aws_security_group.web";
  "aws_internet_gateway.main" -> "aws_vpc.main";
  "aws_route_table.public" -> "aws_internet_gateway.main";
  "aws_route_table_association.public" -> "aws_route_table.public";
  "aws_route_table_association.public" -> "aws_subnet.public";
  "aws_security_group.web" -> "aws_vpc.main";
  "aws_subnet.public" -> "data.aws_availability_zones.available";
  "aws_subnet.public" -> "aws_vpc.main";
  "aws_vpc_security_group_egress_rule.all" -> "aws_security_group.web";
  "aws_vpc_security_group_ingress_rule.http" -> "aws_security_group.web";
…
```

Τα περισσότερα βέλη είναι οι αναφορές που έγραψες. Το βέλος `"aws_instance.web" -> "aws_route_table_association.public"` είναι το `depends_on`. Κοίτα όμως τι λείπει. Ο server αναφέρει το `aws_subnet.public` και το route table αναφέρει το `aws_vpc.main`, αλλά αυτά τα δύο βέλη δεν υπάρχουν. Το `terraform graph` παραλείπει ένα βέλος όταν η ίδια σειρά προκύπτει ήδη από άλλο δρόμο. Ο server περιμένει το association και το association περιμένει το subnet, άρα ο server περιμένει σίγουρα το subnet. Η εικόνα μένει πιο καθαρή χωρίς να αλλάζει τίποτα στη σειρά.

ΚΡΑΤΑ ΑΥΤΟ

Η σειρά του apply βγαίνει από τις αναφορές, όχι από τη σειρά των blocks ή των αρχείων. Το `depends_on` χρειάζεται μόνο όταν ένα resource χρειάζεται ένα άλλο χωρίς να το αναφέρει, όπως ο server που χρειάζεται το route για το internet.

Αλλαγή στο user_data

Άλλαξε τη σελίδα. Πρόσθεσε μια γραμμή στο template.

user_data.sh.tftpl

```
#!/bin/bash
dnf install -y httpd
cat > /var/www/html/index.html <<'HTML'
<h1>${title}</h1>
<p>Δεύτερη έκδοση</p>
HTML
systemctl enable --now httpd
```

Κανένα αρχείο `.tf` δεν άλλαξε. Η `templatefile` διαβάζει όμως το template σε κάθε plan, οπότε το `user_data` έχει νέα τιμή.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
-/+ destroy and then create replacement
…
  # aws_instance.web must be replaced
-/+ resource "aws_instance" "web" {
…
      ~ user_data                              = <<-EOT # forces replacement
…
          + <p>Δεύτερη έκδοση</p>
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
  ~ public_ip   = "54.214.247.179" -> (known after apply)
…
```

Το σύμβολο είναι `-/+` και δίπλα στο `user_data` γράφει `# forces replacement`. Αυτό το έκανε το `user_data_replace_on_change = true`. Ο provider σημειώνει κάθε αλλαγή στο user_data ως λόγο για νέο server, που θα τρέξει το νέο script στη δική του πρώτη εκκίνηση. Η δημόσια IP θα αλλάξει, όπως δείχνει το output `public_ip`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_instance.web: Destroying... [id=i-f6260f341e77e8c90]
…
aws_instance.web: Destruction complete after 11s
aws_instance.web: Creating...
…
aws_instance.web: Creation complete after 10s [id=i-502b064f2d6abbb20]
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
…
```

Πρώτα σβήστηκε ο παλιός server και μετά φτιάχτηκε ο νέος, η προεπιλογή του κεφαλαίου 14. Το δίκτυο και το security group δεν άλλαξαν.

TERMINAL

```
terraform output public_ip
```

ΕΞΟΔΟΣ

```
"54.214.105.202"
```

Η IP είναι άλλη από πριν, γιατί ο server είναι νέος.

Χωρίς το `user_data_replace_on_change`, ή με `false`, το plan θα έλεγε `update in-place` με σύμβολο `~`. Ο provider θα σταματούσε τον server, θα άλλαζε το user_data και θα τον ξεκινούσε ξανά. Το cloud-init όμως τρέχει τα scripts μόνο στην πρώτη εκκίνηση, οπότε το νέο script δεν θα έτρεχε ποτέ και η σελίδα θα έμενε η παλιά. Με το `true` ο κώδικας και ο server δεν αποκλίνουν. Η άσκηση 18.5 δείχνει το plan με `false`.

ΠΑΓΙΔΑ

Το `most_recent = true` έχει μια παρενέργεια. Όταν η Amazon βγάλει νέο image του Amazon Linux, το data source θα βρει νέο ID στο επόμενο plan. Η αλλαγή στο `ami` ενός `aws_instance` σημαίνει αντικατάσταση, οπότε το plan θα θέλει νέο server χωρίς να έχεις αλλάξει γραμμή. Διάβαζε το plan πριν από κάθε apply. Αν θέλεις να αλλάζει ο server μόνο όταν το αποφασίσεις, βάλε `ignore_changes = [ami]` στο `lifecycle` (κεφάλαιο 14) ή δώσε το ID με μεταβλητή.

Το destroy του web

Πριν περάσεις στο module, σβήσε το stack.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_vpc_security_group_egress_rule.all: Destroying... [id=sgr-c05f8b38e22a3f99f]
aws_vpc_security_group_ingress_rule.http: Destroying... [id=sgr-e531dbe942a4452a2]
aws_instance.web: Destroying... [id=i-502b064f2d6abbb20]
aws_vpc_security_group_ingress_rule.http: Destruction complete after 0s
aws_vpc_security_group_egress_rule.all: Destruction complete after 0s
…
aws_instance.web: Destruction complete after 10s
aws_route_table_association.public: Destroying... [id=rtbassoc-da72417445e216324]
aws_security_group.web: Destroying... [id=sg-3fb0c6dcd2595d97c]
aws_route_table_association.public: Destruction complete after 0s
aws_route_table.public: Destroying... [id=rtb-8d8bd6c8476f6857d]
aws_subnet.public: Destroying... [id=subnet-1a6b4cc309482d72a]
aws_security_group.web: Destruction complete after 0s
aws_subnet.public: Destruction complete after 0s
aws_route_table.public: Destruction complete after 0s
aws_internet_gateway.main: Destroying... [id=igw-03bc255f512925e2a]
aws_internet_gateway.main: Destruction complete after 0s
aws_vpc.main: Destroying... [id=vpc-e621ae7c2bd43257c]
```

```
aws_vpc.main: Destruction complete after 0s
…
Destroy complete! Resources: 9 destroyed.
```

Η σειρά είναι η αντίστροφη. Πρώτα φεύγουν ο server και οι δύο κανόνες, που δεν τα χρειάζεται κανείς. Το association, το subnet και το security group περιμένουν να σβηστεί ο server. Το association περιμένει λόγω του `depends_on`, που ισχύει και στο destroy. Το VPC φεύγει τελευταίο. Το AWS δεν δέχεται άλλη σειρά. Ένα subnet δεν σβήνεται όσο έχει server.

Ένα module για το δίκτυο

Το δίκτυο είναι το ίδιο για κάθε server που θα στήσεις. Είναι καλός υποψήφιος για module, με τους κανόνες του κεφαλαίου 15. Φτιάξε νέο φάκελο δίπλα στο `web` και αντίγραψε όσα δεν αλλάζουν.

TERMINAL

```
mkdir ../stack
cd ../stack
cp ../web/versions.tf ../web/outputs.tf ../web/user_data.sh.tftpl .
cp ../web/.terraform.lock.hcl .
```

Το lock file κρατά την ίδια έκδοση με τα ίδια checksums και επιτρέπει στο init να πάρει τον provider από το plugin cache, όπως εξήγησε το κεφάλαιο 17.

Το module ζει στο `modules/network`. Δηλώνει τον provider με ελάχιστη έκδοση και χωρίς block `provider`, όπως έμαθες στο κεφάλαιο 15.

modules/network/versions.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.67.0"
    }
  }
}
```

Οι μεταβλητές είναι όσα αλλάζουν από κλήση σε κλήση. Τα CIDR blocks έχουν defaults, το όνομα όχι.

modules/network/variables.tf

```
variable "name" {
  type        = string
  description = "Το όνομα του δικτύου, για τα tags Name"
}

variable "vpc_cidr" {
  type        = string
  default     = "10.0.0.0/16"
  description = "Το CIDR block του VPC"
}

variable "subnet_cidr" {
  type        = string
  default     = "10.0.1.0/24"
  description = "Το CIDR block του δημόσιου subnet"
}
```

Το `main.tf` του module είναι το `network.tf` του `web`, με μεταβλητές στη θέση των σταθερών τιμών.

modules/network/main.tf

```
data "aws_availability_zones" "available" {
  state = "available"
}

resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true

  tags = {
    Name = var.name
  }
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.subnet_cidr
  availability_zone       = data.aws_availability_zones.available.names[0]
  map_public_ip_on_launch = true

  tags = {
    Name = "${var.name}-public"
  }
}

resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
}

resource "aws_route_table_association" "public" {
```

```
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

Τα outputs είναι η μόνη πόρτα προς τα έξω. Το root χρειάζεται το ID του VPC για το security group και το ID του subnet για τον server.

modules/network/outputs.tf

```
output "vpc_id" {
  value       = aws_vpc.main.id
  description = "Το ID του VPC"
}

output "subnet_id" {
  value       = aws_subnet.public.id
  description = "Το ID του δημόσιου subnet, αφού συνδεθεί με το route table"

  depends_on = [aws_route_table_association.public]
}
```

Το `depends_on` του output (κεφάλαιο 8) είναι το `depends_on` του server, που μετακόμισε στο module. Το root δεν βλέπει το association, άρα δεν μπορεί να το γράψει σε δικό του `depends_on`. Το module όμως εγγυάται ότι το `subnet_id` φτάνει στο root μόνο όταν το subnet έχει route προς το internet. Όποιος χρησιμοποιεί το output περιμένει αυτόματα. Ένα `depends_on = [module.network]` στον server (κεφάλαιο 15) θα δούλευε κι αυτό, αλλά θα περίμενε κάθε resource του module.

Στο root μένουν η κλήση, το security group και ο server.

main.tf

```
module "network" {
  source = "./modules/network"

  name = "tfbook-stack"
}
```

security.tf

```
resource "aws_security_group" "web" {
  name        = "tfbook-stack-web"
  description = "HTTP from anywhere"
  vpc_id      = module.network.vpc_id
}

resource "aws_vpc_security_group_ingress_rule" "http" {
  security_group_id = aws_security_group.web.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "tcp"
  from_port         = 80
  to_port           = 80
}

resource "aws_vpc_security_group_egress_rule" "all" {
  security_group_id = aws_security_group.web.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "-1"
}
```

server.tf

```
data "aws_ami" "al2023" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["al2023-ami-2023.*-kernel-6.1-x86_64"]
  }
}

resource "aws_instance" "web" {
  ami                    = data.aws_ami.al2023.id
  instance_type          = "t3.micro"
  subnet_id              = module.network.subnet_id
  vpc_security_group_ids = [aws_security_group.web.id]

  user_data = templatefile("${path.module}/user_data.sh.tftpl", {
    title = "Γεια σου από το module"
  })
  user_data_replace_on_change = true

  tags = {
    Name = "tfbook-stack-web"
  }
}
```

Οι αναφορές `module.network.vpc_id` και `module.network.subnet_id` είναι τα δύο outputs. Ο server δεν έχει πια `depends_on`. Το `outputs.tf` που αντέγραψες δουλεύει όπως είναι, γιατί αναφέρει μόνο το `aws_instance.web`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
Initializing modules...
- network in modules/network
…
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v6.67.0...
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
…
```

Το init βρήκε το module στο `modules/network`. Τον provider τον πήρε από το lock file, με την έκδοση που είχε ήδη διαλέξει το `web`.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_instance.web will be created
…
  # aws_security_group.web will be created
…
  # aws_vpc_security_group_egress_rule.all will be created
…
  # aws_vpc_security_group_ingress_rule.http will be created
…
  # module.network.aws_internet_gateway.main will be created
…
  # module.network.aws_route_table.public will be created
…
  # module.network.aws_route_table_association.public will be created
…
  # module.network.aws_subnet.public will be created
…
  # module.network.aws_vpc.main will be created
…
Plan: 9 to add, 0 to change, 0 to destroy.
…
```

Τα resources του module έχουν διευθύνσεις με πρόθεμα `module.network.`. Η περίληψη είναι ξανά εννέα resources. Το module δεν πρόσθεσε τίποτα στο AWS, μόνο οργάνωση στον κώδικα.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
module.network.aws_route_table_association.public: Creating...
module.network.aws_route_table_association.public: Creation complete after 0s [id=rtbassoc-c831624672ac3c7d8]
aws_instance.web: Creating...
…
aws_instance.web: Creation complete after 12s [id=i-d4a66dd2875e15342]
…
Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
…
```

Ο server ξεκίνησε αμέσως μετά το association του module. Η εξάρτηση πέρασε από το output, χωρίς ο server να ξέρει ότι υπάρχει association.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
data.aws_ami.al2023
aws_instance.web
aws_security_group.web
aws_vpc_security_group_egress_rule.all
aws_vpc_security_group_ingress_rule.http
module.network.data.aws_availability_zones.available
module.network.aws_internet_gateway.main
module.network.aws_route_table.public
module.network.aws_route_table_association.public
module.network.aws_subnet.public
module.network.aws_vpc.main
```

Το data source του module έχει κι αυτό το πρόθεμα, `module.network.data.aws_availability_zones.available`. Αν το root χρειαζόταν και το CIDR block του subnet, η έκφραση `module.network.aws_subnet.public.cidr_block` θα έβγαζε `Unsupported attribute`, όπως στο κεφάλαιο 15. Το module θα έπρεπε να το δώσει με νέο output.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Από ένα child module διαβάζεις μόνο outputs, με τη μορφή `module.ONOMA.OUTPUT`. Μια αναφορά σε output δημιουργεί implicit dependency όπως κάθε αναφορά. Το resource που τη χρησιμοποιεί περιμένει ό,τι χρειάζεται το output, ακόμη και resources μέσα στο module που δεν βλέπει.

TERMINAL

```
terraform output url
```

ΕΞΟΔΟΣ

```
"http://54.214.142.58"
```

Καθάρισμα

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 9 destroyed.
```

Έλεγξε από τη μεριά του AWS ότι δεν τρέχει κανένας server.

TERMINAL

```
aws ec2 describe-instances --filters Name=instance-state-name,Values=running \
  --query 'Reservations[].Instances[].InstanceId'
```

ΕΞΟΔΟΣ

```
[]
```

Η κενή λίστα σημαίνει ότι δεν έμεινε κανένας server σε λειτουργία.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Το VPC, το subnet, το internet gateway, το route table και το security group δεν χρεώνονται. Χρεώνονται ο server, ο δίσκος του και η δημόσια IPv4. Με τον τιμοκατάλογο του 2026 ένα `t3.micro` με Linux στη Φρανκφούρτη κοστίζει 0,012 δολάρια την ώρα και κάθε δημόσια IPv4 0,005 δολάρια την ώρα. Λίγα, αλλά ένας ξεχασμένος server χρεώνεται κάθε ώρα του μήνα. Κάνε destroy στο τέλος κάθε παραδείγματος και κάθε άσκησης και έλεγξε με το `describe-instances`. Ένας σβησμένος server φαίνεται για λίγη ώρα ως `terminated`, χωρίς χρέωση. Κάθε λογαριασμός έχει επίσης από προεπιλογή όριο πέντε VPCs ανά region, που τα ξεχασμένα stacks εξαντλούν γρήγορα.

Ασκήσεις

Οι ασκήσεις ξεκινούν από αντίγραφο του `web` ή του `stack`, όπως λέει η εκφώνηση. Το αντίγραφο έχει τα αρχεία `.tf`, το template, το `modules/network` και το lock file, χωρίς state και χωρίς `.terraform`. Χρησιμοποίησε `-auto-approve` όπου θέλεις. Στο πραγματικό AWS κάνε destroy στο τέλος κάθε άσκησης.

ΑΣΚΗΣΗ 18.1 Χωρίς depends_on

Ξεκίνα από το `web`. Σβήσε το `depends_on` από τον server. Πριν τρέξεις οτιδήποτε, γράψε σε χαρτί ποια βέλη θα ξεκινούν από το `aws_instance.web` στο `terraform graph`. Έλεγξε κρατώντας μόνο αυτές τις γραμμές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  "aws_instance.web" -> "data.aws_ami.al2023";
  "aws_instance.web" -> "aws_security_group.web";
  "aws_instance.web" -> "aws_subnet.public";
…
```

ΑΣΚΗΣΗ 18.2 Ζώνες και εικόνα

Ξεκίνα από το `web`. Πρόσθεσε το αρχείο `lookup.tf` με δύο outputs. Το `zones` δίνει όλα τα ονόματα των availability zones και το `ami_name` το όνομα του image που βρήκε το `data.aws_ami.al2023`. Τρέξε μόνο plan και βρες τις τιμές στο `Changes to Outputs`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  + ami_name    = "al2023-ami-2023.12.20260727.0-kernel-6.1-x86_64"
…
  + zones       = [
      + "eu-central-1a",
      + "eu-central-1b",
      + "eu-central-1c",
…
```

ΑΣΚΗΣΗ 18.3 Χωρίς δημόσια IP

Ξεκίνα από το `web`. Άλλαξε το `map_public_ip_on_launch` του subnet σε `false`. Κάνε apply και τύπωσε το output `public_ip`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
""
```

ΑΣΚΗΣΗ 18.4 HTTPS και μεγαλύτερος server

Ξεκίνα από το `web` και κάνε apply. Μετά κάνε δύο αλλαγές. Πρόσθεσε στο αρχείο `https.tf` κανόνα εισόδου με όνομα `https` για τη θύρα 443 από παντού. Άλλαξε το `instance_type` σε `t3.small`. Πριν τρέξεις plan, γράψε σε χαρτί ποιο σύμβολο περιμένεις για κάθε resource και τι θα λέει η περίληψη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_instance.web will be updated in-place
…
      ~ instance_type                        = "t3.micro" -> "t3.small"
…
  # aws_vpc_security_group_ingress_rule.https will be created
…
Plan: 1 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 18.5 user_data χωρίς αντικατάσταση

Ξεκίνα από το `web` και κάνε apply. Άλλαξε μετά το `user_data_replace_on_change` σε `false` και πρόσθεσε στο template τη γραμμή `<p>Τρίτη έκδοση</p>` κάτω από τη δεύτερη. Τρέξε plan. Τι σύμβολο έχει ο server αυτή τη φορά;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_instance.web will be updated in-place
…
      ~ user_data                             = <<-EOT
…
          + <p>Τρίτη έκδοση</p>
…
      ~ user_data_replace_on_change           = true -> false
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 18.6 Η ζώνη στη σελίδα

Ξεκίνα από το `stack`. Η σελίδα πρέπει να γράφει και την availability zone του server, σε μια γραμμή `<p>Zone ...</p>` κάτω από τον τίτλο. Πρόσθεσε στο module output `availability_zone`, δώσε το στο template ως μεταβλητή `zone` και τρέξε plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
            <p>Zone eu-central-1a</p>
…
Plan: 9 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 18.7 Δύο δίκτυα από ένα module

Ξεκίνα από το `stack`. Πρόσθεσε στο `main.tf` δεύτερη κλήση του module με όνομα κλήσης `test`, όνομα δικτύου `tfbook-test`, VPC `10.1.0.0/16` και subnet `10.1.1.0/24`. Τρέξε plan και κράτα μόνο τα resources της νέας κλήσης και την περίληψη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # module.test.aws_internet_gateway.main will be created
…
  # module.test.aws_route_table.public will be created
…
  # module.test.aws_route_table_association.public will be created
…
  # module.test.aws_subnet.public will be created
…
  # module.test.aws_vpc.main will be created
…
Plan: 14 to add, 0 to change, 0 to destroy.
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 18.1

Μια configuration έχει ένα `aws_vpc.main`, ένα `aws_subnet.public` και ένα `aws_security_group.web`. Το subnet και το security group αναφέρουν το `aws_vpc.main.id`. Ένα `aws_instance.web` αναφέρει το `aws_subnet.public.id` και το `aws_security_group.web.id`. Ποια πρόταση περιγράφει σωστά τη σειρά του apply;

a. Το Terraform φτιάχνει τα resources με τη σειρά που εμφανίζονται στα αρχεία.
b. Πρώτα το VPC. Μετά το subnet και το security group, πιθανόν ταυτόχρονα. Τελευταίο ο server.
c. Ο server και το security group φτιάχνονται ταυτόχρονα, γιατί το subnet δεν αναφέρει το security group.
d. Η σειρά δεν είναι εγγυημένη χωρίς `depends_on` στον server.

ΕΡΩΤΗΣΗ 18.2

Ένα `data "aws_ami"` έχει `owners = ["amazon"]` και ένα `filter` στο όνομα με `*`. Δεν έχει `most_recent`. Το φίλτρο ταιριάζει σε τρία images. Τι κάνει το `terraform plan`;

a. Διαλέγει το πρώτο image με αλφαβητική σειρά.
b. Διαλέγει το νεότερο image.
c. Σταματά με σφάλμα που ζητά πιο συγκεκριμένο φίλτρο ή `most_recent = true`.
d. Επιστρέφει λίστα με τα τρία IDs στο attribute `id`.

ΕΡΩΤΗΣΗ 18.3

Ένα child module με όνομα κλήσης `network` περιέχει το resource `aws_subnet.public` και δηλώνει `output "subnet_id"` με τιμή `aws_subnet.public.id`. Ποια έκφραση δίνει το ID του subnet στο root module;

a. `module.network.aws_subnet.public.id`
b. `module.network.subnet_id`
c. `output.network.subnet_id`
d. `var.network.subnet_id`

ΕΡΩΤΗΣΗ 18.4

Ένα `aws_instance` έχει `user_data = templatefile(...)` και `user_data_replace_on_change = true`. Αλλάζεις μόνο το αρχείο του template. Τι δείχνει το επόμενο plan για τον server;

a. Τίποτα, γιατί κανένα αρχείο `.tf` δεν άλλαξε.
b. Αντικατάσταση με `-/+` και `# forces replacement` δίπλα στο `user_data`.
c. Update in place με `~` στον ίδιο server.
d. Σφάλμα, γιατί το user_data δεν αλλάζει μετά τη δημιουργία.

ΕΡΩΤΗΣΗ 18.5

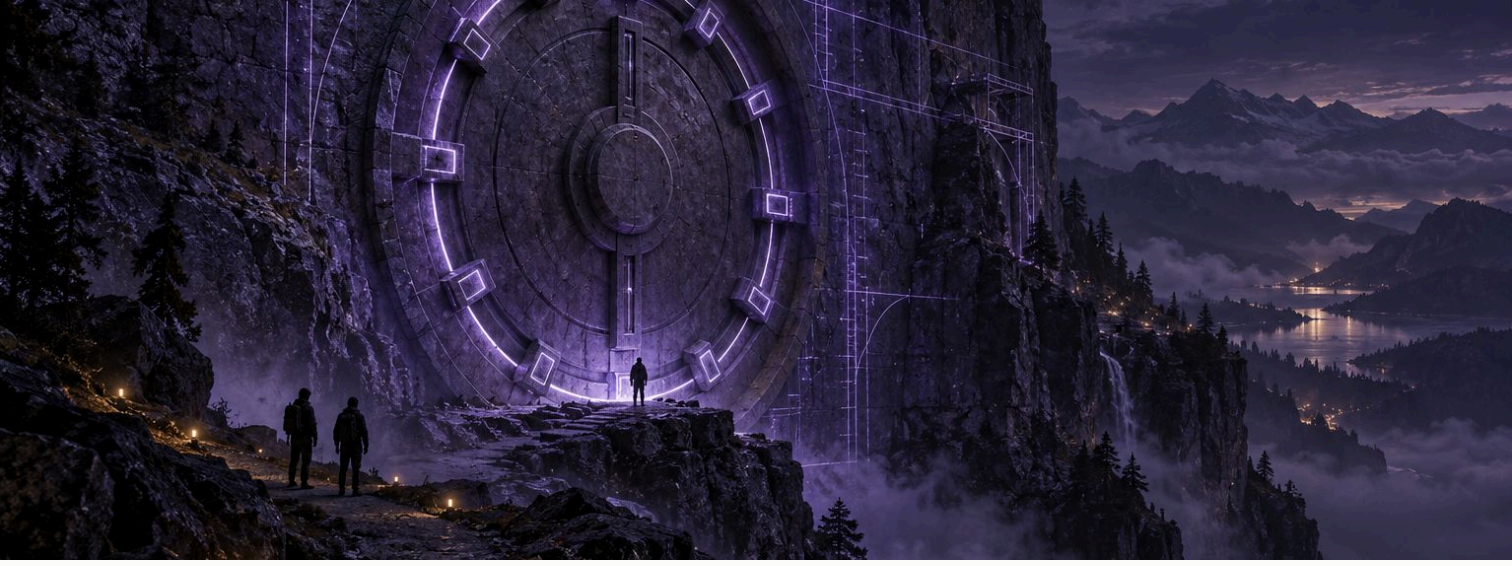
Ποιες δύο προτάσεις για το `depends_on` είναι σωστές; Επίλεξε δύο.

a. Χρειάζεται όταν ένα resource χρειάζεται ένα άλλο χωρίς να το αναφέρει σε καμία έκφραση.
b. Μπορεί να αναφέρει ολόκληρο module, οπότε το resource περιμένει όλα τα resources του module.
c. Πρέπει να γράφεται για κάθε αναφορά, αλλιώς η σειρά δεν είναι εγγυημένη.
d. Κάνει το Terraform να δουλεύει ένα resource τη φορά.

ΕΡΩΤΗΣΗ 18.6

Σωστό ή λάθος; Το plan μιας νέας configuration δείχνει τη δημόσια IP που θα πάρει ένα `aws_instance`, ώστε να τη βάλεις σε κανόνα firewall πριν από το apply.

a. Σωστό
b. Λάθος



ΚΕΦΑΛΑΙΟ 19

Backends, remote state και locking

Ως τώρα το state ζούσε σε ένα αρχείο δίπλα στον κώδικα. Εδώ το μεταφέρεις σε ένα S3 bucket για όλη την ομάδα. Βλέπεις το lock στην πράξη, ξεκλειδώνεις ένα lock που έμεινε ορφανό και διαβάζεις τα outputs μιας configuration από μια άλλη.

Το περιβάλλον

Τα παραδείγματα τρέχουν στο περιβάλλον του κεφαλαίου 17. Το Moto τρέχει σε Docker και στο terminal σου είναι ορισμένες οι μεταβλητές `AWS_ENDPOINT_URL`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` και `AWS_REGION`. Έλεγξε ότι το `aws` CLI μιλά με το Moto.

TERMINAL

```
echo $AWS_ENDPOINT_URL
aws sts get-caller-identity
```

ΕΞΟΔΟΣ

```
http://localhost:5050
{
    "UserId": "AKIAIOSFODNN7EXAMPLE",
    "Account": "123456789012",
    "Arn": "arn:aws:sts::123456789012:user/moto"
}
```

Ο λογαριασμός `123456789012` είναι ο ψεύτικος λογαριασμός του Moto. Αν δεις άλλο αριθμό, μιλάς με πραγματικό AWS. Γύρνα τότε στο κεφάλαιο 17 πριν συνεχίσεις.

Γιατί δεν φτάνει το τοπικό state

Στο κεφάλαιο 4 το state ήταν το `terraform.tfstate` του φακέλου εργασίας. Για έναν άνθρωπο σε ένα laptop αρκεί. Με δεύτερο άνθρωπο ή με ένα pipeline εμφανίζονται τέσσερα προβλήματα.

- **Κοινή πρόσβαση.** Ο συνάδελφός σου δεν έχει το state σου. Το δικό του apply θα προσπαθήσει να ξαναφτιάξει όσα έφτιαξες. Με το state σε Git αρκεί ένα ξεχασμένο pull για να αποκλίνουν τα αντίγραφα.
- **Ταυτόχρονες αλλαγές.** Το lock του local backend δεν φαίνεται από άλλο υπολογιστή. Αν δύο apply τρέξουν μαζί, το state που γράφεται τελευταίο σβήνει ό,τι έγραψε το άλλο.
- **Απώλεια.** Ένας χαμένος δίσκος και το Terraform ξεχνά όσα διαχειρίζεται, ενώ τα resources μένουν στο cloud.
- **Μυστικά.** Το state κρατά attributes σε απλό κείμενο. Κάθε αντίγραφό του είναι και αντίγραφο των μυστικών.

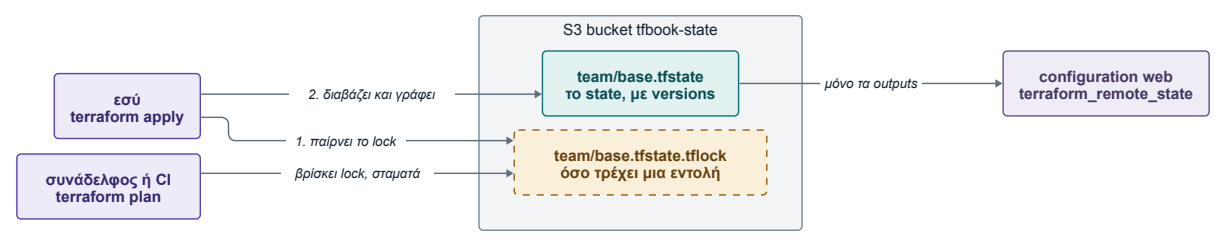
Χρειάζεσαι ένα κοινό σημείο με κοινό lock, ιστορικό εκδόσεων και έλεγχο πρόσβασης. Αυτό δίνει ένα remote backend.

Τι είναι backend

Το **backend** αποφασίζει πού αποθηκεύεται το state και πώς κλειδώνεται. Χωρίς ρύθμιση ισχύει το local backend του κεφαλαίου 4. Ένα **remote backend** κρατά το state σε μια υπηρεσία δικτύου, όπως ένα S3 bucket ή μια βάση PostgreSQL. Σε κάθε εντολή το Terraform κατεβάζει το state, το κρατά στη μνήμη και ανεβάζει τη νέα έκδοση όταν αλλάξει. Στον φάκελο εργασίας δεν γράφεται αντίγραφο.

Το backend ρυθμίζεται με ένα block `backend` μέσα στο block `terraform`, με τον τύπο του backend ως label. Μία configuration έχει το πολύ ένα backend και κάθε αλλαγή του θέλει `terraform init`. Τα backends είναι ενσωματωμένα στο Terraform. Δεν είναι providers και δεν δηλώνονται στο `required_providers`. Γι' αυτό τα παραδείγματα του κεφαλαίου κρατούν το state τους στο S3 χωρίς να κατεβάσουν τον AWS provider.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
>
> Το local backend είναι η προεπιλογή όταν δεν υπάρχει block `backend` ή `cloud`. Ένα remote backend δίνει κοινή πρόσβαση στο state, lock για όλη την ομάδα και, ανάλογα με την υπηρεσία, κρυπτογράφηση και ιστορικό. Τα backends είναι μέρος του Terraform, όχι providers.



Σχήμα 19.1 · Το S3 bucket κρατά το state και το lock. Όποια εντολή πάρει πρώτη το lock διαβάζει και γράφει το state, οι υπόλοιπες σταματούν. Μια άλλη configuration διαβάζει μόνο τα outputs του.

Το bucket του state

Το S3 backend δεν φτιάχνει το bucket του. Το bucket πρέπει να υπάρχει πριν από το `init`, άρα δεν μπορεί να το φτιάξει η configuration που θα κρατά εκεί το state της. Το φτιάχνεις μία φορά με το `aws` CLI ή με μια μικρή ξεχωριστή configuration με local state.

TERMINAL

```
aws s3api create-bucket --bucket tfbook-state \
  --create-bucket-configuration LocationConstraint=eu-central-1
aws s3api put-bucket-versioning --bucket tfbook-state \
  --versioning-configuration Status=Enabled
aws s3api get-bucket-versioning --bucket tfbook-state
```

ΕΞΟΔΟΣ

```
{
    "Location": "/tfbook-state"
}
{
    "Status": "Enabled"
}
```

Έξω από το `us-east-1` το S3 θέλει το region και στο `LocationConstraint`, αλλιώς απαντά `IllegalLocationConstraintException`. Η δεύτερη εντολή ενεργοποιεί το versioning και η τρίτη το επιβεβαιώνει. Με versioning κάθε apply γράφει νέα έκδοση του state και οι παλιές μένουν. Αν ένα λάθος χαλάσει το state, η προηγούμενη έκδοση είναι ακόμη εκεί.

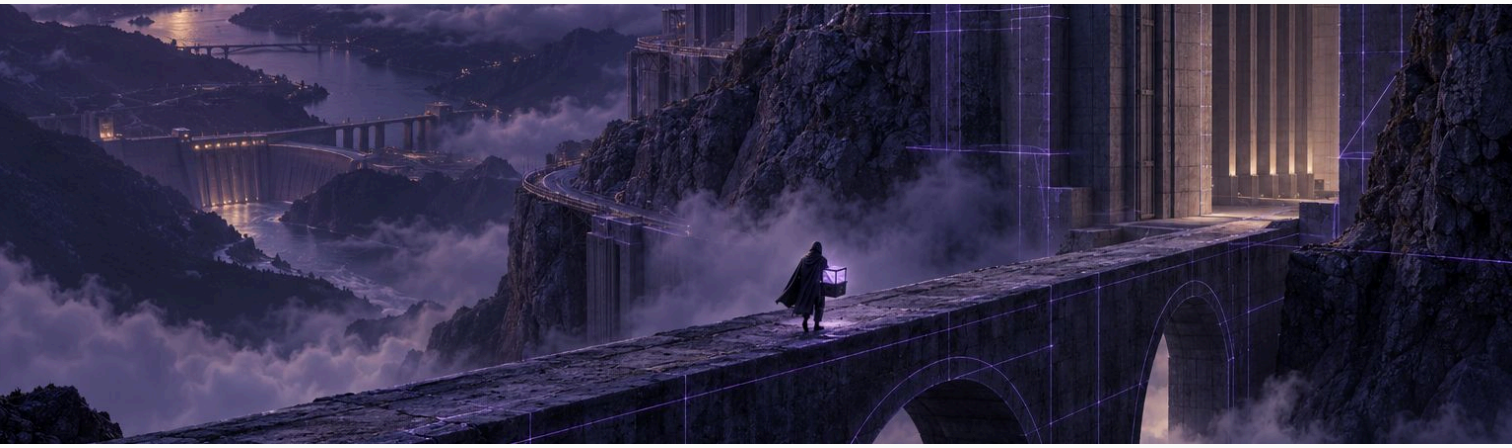
Ο δεύτερος τρόπος μοιάζει με το παρακάτω. Το `prevent_destroy` του κεφαλαίου 14 εμποδίζει ένα λάθος `destroy` να σβήσει τα states όλης της ομάδας.

HCL

```
resource "aws_s3_bucket" "state" {
  bucket = "tfbook-state"

  lifecycle {
    prevent_destroy = true
  }
}

resource "aws_s3_bucket_versioning" "state" {
  bucket = aws_s3_bucket.state.id

  versioning_configuration {
    status = "Enabled"
  }
}
```

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Τα ονόματα των buckets είναι μοναδικά σε όλο το AWS, οπότε το `tfbook-state` είναι σχεδόν σίγουρα πιασμένο. Βάλε στο όνομα την ομάδα ή τον λογαριασμό σου. Τα νέα buckets έχουν από προεπιλογή κρυπτογράφηση και μπλοκαρισμένη δημόσια πρόσβαση. Δώσε δικαιώματα μόνο σε όσους τρέχουν Terraform. Οι παλιές εκδόσεις μένουν και χρεώνονται μέχρι να τις σβήσεις, για παράδειγμα με έναν κανόνα lifecycle. Στο Moto το bucket χάνεται όταν σταματήσει το container.



Από το local state στο S3

Ξεκίνα από ένα project με τοπικό state. Η configuration `base` φτιάχνει ένα τυχαίο πρόθεμα ονομάτων για όλα τα projects της ομάδας.

TERMINAL

```
mkdir base
cd base
```

main.tf

```
terraform {
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "prefix" {
  length = 2
}

output "prefix" {
  value = random_pet.prefix.id
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
random_pet.prefix: Creation complete after 0s [id=enhanced-cowbird]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
prefix = "enhanced-cowbird"
```

TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
```

Το state είναι στο `terraform.tfstate`. Γράψε το backend σε δεύτερο αρχείο. Το Terraform ενώνει τα blocks `terraform` όλων των αρχείων, οπότε αυτό επιτρέπεται.

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "base/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

- `bucket` είναι το bucket που μόλις έφτιαξες.
- `key` είναι το όνομα του αντικειμένου στο bucket. Κάθε configuration θέλει δικό της key, αλλιώς οι configurations γράφουν η μία πάνω στην άλλη.
- `region` είναι το region του bucket.
- `use_lockfile` ενεργοποιεί το lock. Χωρίς αυτό το S3 backend δεν κλειδώνει, εκτός αν έχει το παλιό `dynamodb_table` που θα δεις παρακάτω.

Το backend βρίσκει credentials όπως ο AWS provider, από τις ίδιες μεταβλητές και τα ίδια αρχεία του κεφαλαίου 17. Το block `provider "aws"` όμως δεν το διαβάζει. Το backend άλλαξε, οπότε τρέχεις `init`. Το `-migrate-state` ζητά μεταφορά του υπάρχοντος state.

ΣΗΜΕΙΩΣΗ

Το S3 backend βάζει το όνομα του bucket στο hostname, οπότε με το Moto μιλά με το `tfbook-state.localhost`. Αν το σύστημά σου δεν λύνει τέτοια ονόματα και το `init` αποτύχει να συνδεθεί, πρόσθεσε `use_path_style = true` στο block `backend`. Το ίδιο ισχύει για το `config` του `terraform_remote_state` που θα δεις παρακάτω. Στο πραγματικό AWS δεν χρειάζεται.

TERMINAL

```
terraform init -migrate-state
```

ΕΞΟΔΟΣ

```
Initializing the backend...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend to the
  newly configured "s3" backend. No existing state was found in the newly
  configured "s3" backend. Do you want to copy this state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes


Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
…
```

Το Terraform βρήκε τοπικό state και άδειο S3 και ρώτησε αν θα αντιγράψει. Με `no` θα ξεκινούσε με άδειο state και τα resources σου θα έμεναν χωρίς state. Ένα σκέτο `init` κάνει την ίδια ερώτηση την πρώτη φορά, αλλά το `-migrate-state` λέει ρητά τι θέλεις. Σε scripts το `-force-copy` απαντά `yes` μόνο του.

TERMINAL

```
aws s3 ls s3://tfbook-state --recursive
wc -c terraform.tfstate terraform.tfstate.backup
jq -r .lineage terraform.tfstate.backup
```

ΕΞΟΔΟΣ

```
2026-10-08 05:28:42        775 base/terraform.tfstate
       0 terraform.tfstate
     775 terraform.tfstate.backup
     775 total
18e1fc92-a46c-bc16-7f8e-58e1bd43b580
```

Το state είναι πλέον το `base/terraform.tfstate`. Το τοπικό αρχείο άδειασε στα 0 bytes και το παλιό περιεχόμενο έμεινε στο backup. Το Terraform δεν τα διαβάζει πια. Σβήσ’ τα, για να μη τα περάσει κανείς για το τρέχον state.

TERMINAL

```
rm terraform.tfstate terraform.tfstate.backup
terraform plan
```

ΕΞΟΔΟΣ

```
random_pet.prefix: Refreshing state... [id=enhanced-cowbird]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το plan βρήκε το `random_pet.prefix` χωρίς κανένα τοπικό αρχείο. Το διάβασε από το S3. Το remote state το βλέπεις όπως είναι με `terraform state pull`.

TERMINAL

```
terraform state pull | jq '{serial, lineage, outputs}'
```

ΕΞΟΔΟΣ

```
{
  "serial": 1,
  "lineage": "2fe1211c-b737-6714-e8b0-49973f27a5ca",
  "outputs": {
    "prefix": {
      "value": "enhanced-cowbird",
      "type": "string"
    }
  }
}
```

Τα outputs ήρθαν από το τοπικό state. Το `lineage` όμως διαφέρει από αυτό του backup και το `serial` ξεκίνησε από το 1. Η μεταφορά δεν ανεβάζει το αρχείο αυτούσιο. Γράφει τα ίδια resources και outputs σε ένα νέο state, που παίρνει δικό του `lineage` όπως κάθε νέο state του κεφαλαίου 4. Η 1.12 της εξέτασης κάνει το ίδιο σε κάθε μεταφορά backend. Το αντίθετο του `state pull`, το `state push`, είναι θέμα του κεφαλαίου 21.

Ένα ακόμη apply, με το `-replace` του κεφαλαίου 3, δείχνει το versioning.

TERMINAL

```
terraform apply -auto-approve -replace=random_pet.prefix
```

ΕΞΟΔΟΣ

```
…
random_pet.prefix: Creation complete after 0s [id=viable-piglet]
…
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
…
prefix = "viable-piglet"
```

TERMINAL

```
aws s3api list-object-versions --bucket tfbook-state \
  --query "length(Versions[?Key=='base/terraform.tfstate'])"
```

ΕΞΟΔΟΣ

```
2
```

Το bucket κρατά δύο εκδόσεις του state. Η παλιά κατεβαίνει με `aws s3api get-object --version-id`, αν χρειαστεί να γυρίσεις πίσω.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το remote state ζει σε ένα σημείο για όλους. Ο φάκελος εργασίας κρατά μόνο τη ρύθμιση του backend στο `.terraform`. Το `state pull`, το `state list` και το `show` δουλεύουν όπως με το local backend.

Το lock του S3 και το παλιό DynamoDB

Με `use_lockfile` το Terraform κλειδώνει γράφοντας δίπλα στο state ένα αντικείμενο με κατάληξη `.tflock`. Η εγγραφή είναι conditional write, που το S3 τη δέχεται μόνο αν το αντικείμενο δεν υπάρχει. Από δύο εντολές μόνο η μία πετυχαίνει. Στο τέλος η εντολή σβήνει το `.tflock`.

Πριν από αυτό το lock γινόταν σε πίνακα DynamoDB, με το argument `dynamodb_table`, που θα το βρεις σε πολλά projects. Το `use_lockfile` πρωτοεμφανίστηκε στην 1.10 και έγινε σταθερό στην 1.11. Από την 1.11 η τεκμηρίωση δηλώνει το `dynamodb_table` deprecated. Θα αφαιρεθεί σε επόμενη έκδοση.

TERMINAL

```
cd ..
mkdir legacy
cd legacy
```

backend.tf

```
terraform {
  backend "s3" {
    bucket         = "tfbook-state"
    key            = "legacy/terraform.tfstate"
    region         = "eu-central-1"
    dynamodb_table = "tfbook-locks"
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
│ Warning: Deprecated Parameter
…
│    6:     dynamodb_table = "tfbook-locks"
…
│ The parameter "dynamodb_table" is deprecated. Use parameter "use_lockfile"
│ instead.
…
Terraform has been successfully initialized!
…
```

Το init πέτυχε με προειδοποίηση `Deprecated Parameter`. Την προειδοποίηση την τυπώνει το Terraform από την 1.15. Η 1.12 της εξέτασης δέχεται το `dynamodb_table` χωρίς καμία προειδοποίηση. Ο πίνακας `tfbook-locks` δεν υπάρχει καν, αλλά το init δεν κλειδώνει και δεν το πρόσεξε. Το πρώτο plan θα σταματούσε με `Error acquiring the state lock`.

Σε ένα παλιό project πρόσθεσε `use_lockfile = true` δίπλα στο `dynamodb_table` και τρέξε `terraform init -reconfigure`. Η θέση του state δεν αλλάζει, οπότε δεν χρειάζεται μεταφορά. Η επόμενη ενότητα εξηγεί τη σημαία. Με τα δύο μαζί το Terraform παίρνει και τα δύο locks, οπότε προστατεύει και όσους τρέχουν ακόμη παλιότερη έκδοση. Σβήσε το `dynamodb_table` όταν δεν το χρειάζεται πια κανείς.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Στην 1.12 της εξέτασης ισχύουν και τα δύο. Το `use_lockfile` είναι ο τρόπος που προτείνεται και το `dynamodb_table` είναι deprecated, αν και η 1.12 δεν το λέει στην έξοδο. Τα δύο συνυπάρχουν όσο διαρκεί η μετάβαση. Χωρίς κανένα από τα δύο το S3 backend δεν έχει lock.

Αλλαγή backend με -migrate-state και -reconfigure

Κάθε αλλαγή στο block `backend`, ακόμη και σε ένα key, θέλει νέο `init`. Η ομάδα αποφασίζει να κρατά τα states κάτω από το `team/`. Γύρνα στο `base` και άλλαξε το key.

TERMINAL

```
cd ../base
```

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "team/base.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

╷
│ Error: Backend configuration changed
│
│ A change in the backend configuration has been detected, which may require
│ migrating existing state.
│
│ If you wish to attempt automatic migration of the state, run "terraform
│ init -migrate-state".
│ If you wish to store the current configuration with no changes to the
│ state, run "terraform init -reconfigure".
╵
```

Το init δεν μαντεύει τι θέλεις και προτείνει δύο σημαίες.

- Το `-migrate-state` αντιγράφει το state από την παλιά θέση στη νέα, αφού ρωτήσει.
- Το `-reconfigure` ξεχνά την παλιά ρύθμιση και δουλεύει με ό,τι βρει στη νέα θέση, χωρίς ερώτηση και χωρίς αντιγραφή.

Εδώ θέλεις μεταφορά.

TERMINAL

```
terraform init -migrate-state
```

ΕΞΟΔΟΣ

```
Initializing the backend...
Backend configuration changed!

Terraform has detected that the configuration specified for the backend
has changed. Terraform will now check for existing state in the backends.
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "s3" backend to the
  newly configured "s3" backend. No existing state was found in the newly
  configured "s3" backend. Do you want to copy this state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes


Successfully configured the backend "s3"! Terraform will automatically
…
```

TERMINAL

```
aws s3 ls s3://tfbook-state --recursive
```

ΕΞΟΔΟΣ

```
2026-10-08 05:28:44        769 base/terraform.tfstate
2026-10-08 05:28:45        769 team/base.tfstate
```

Η μεταφορά αντιγράφει και δεν μετακινεί, οπότε το παλιό αντικείμενο έμεινε. Σβήσ’ το όταν βεβαιωθείς ότι όλα δουλεύουν.

TERMINAL

```
aws s3 rm s3://tfbook-state/base/terraform.tfstate
terraform plan
```

ΕΞΟΔΟΣ

```
…
random_pet.prefix: Refreshing state... [id=viable-piglet]
…
No changes. Your infrastructure matches the configuration.
…
```

Δες τώρα το `-reconfigure` σε λάθος χρήση. Ένας συνάδελφος γράφει το key `team/bsae.tfstate` αντί για `team/base.tfstate`. Μετά τρέχει init με `-reconfigure` για να μη δει την ερώτηση.

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "team/bsae.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init -reconfigure
terraform plan
```

ΕΞΟΔΟΣ

```
…
Successfully configured the backend "s3"! Terraform will automatically
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
```

Το init πέτυχε χωρίς ερώτηση. Το νέο key είναι άδειο, οπότε το plan θέλει να φτιάξει ξανά το `random_pet.prefix` με `1 to add`. Με πραγματικά resources θα είχες δεύτερο αντίγραφο της υποδομής ή σφάλματα για ονόματα που υπάρχουν. Διόρθωσε το key και ξανατρέξε το init.

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "team/base.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init -reconfigure
terraform plan
```

ΕΞΟΔΟΣ

```
…
Successfully configured the backend "s3"! Terraform will automatically
…
No changes. Your infrastructure matches the configuration.
…
```

Εδώ το `-reconfigure` είναι σωστό. Η νέα θέση έχει ήδη το σωστό state, όπως όταν κάποιος έχει κάνει ήδη τη μεταφορά ή όταν αλλάζεις περιβάλλον με partial configuration.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Μετά από κάθε αλλαγή στο block `backend` τρέχεις `terraform init`. Το `-migrate-state` αντιγράφει το υπάρχον state στο νέο backend. Το `-reconfigure` αγνοεί την προηγούμενη ρύθμιση και δεν μεταφέρει τίποτα. Από το local σε ένα remote backend και πίσω περνάς με `init -migrate-state`.

Μεταβλητές στο backend

Θα ήθελες ο ίδιος κώδικας να γράφει σε διαφορετικό key ανά περιβάλλον. Η πρώτη ιδέα είναι μια μεταβλητή ή ένα local. Δοκίμασέ το στη configuration `web`.

TERMINAL

```
cd ..
mkdir web
cd web
```

backend.tf

```
locals {
  state_bucket = "tfbook-state"
}

variable "state_key" {
  type    = string
  default = "web/terraform.tfstate"
}

terraform {
  backend "s3" {
    bucket       = local.state_bucket
    key          = var.state_key
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

╷
│ Error: Variables not allowed
│
│   on backend.tf line 12, in terraform:
│   12:     bucket       = local.state_bucket
│
│ Variables may not be used here.
╵

╷
│ Error: Variables not allowed
│
│   on backend.tf line 13, in terraform:
│   13:     key          = var.state_key
│
│ Variables may not be used here.
╵
```

Δύο σφάλματα `Variables not allowed`, ένα για το local και ένα για τη μεταβλητή. Το backend ρυθμίζεται στο `init`, πριν το Terraform υπολογίσει οτιδήποτε. Πρέπει να ξέρει πού είναι το state πριν από όλα τα άλλα. Γι' αυτό το block `backend` δέχεται μόνο σταθερές τιμές. Όχι `var`, όχι `local`, όχι attributes από resources ή data sources.

Partial configuration

Η λύση είναι η **partial configuration**. Στο block αφήνεις όσα είναι ίδια παντού και δίνεις τα υπόλοιπα στο `terraform init` με `-backend-config`, ως αρχείο ή ως `key=value`.

backend.tf

```
terraform {
  backend "s3" {
    use_lockfile = true
  }
}
```

backend.hcl

```
bucket = "tfbook-state"
region = "eu-central-1"
```

Το `backend.hcl` έχει σκέτα arguments, χωρίς block. Το όνομα είναι σύμβαση. Το Terraform δεν το διαβάζει μόνο του, γιατί δεν τελειώνει σε `.tf`. Εδώ το αρχείο κρατά τα κοινά της ομάδας και η γραμμή εντολών το key του project.

TERMINAL

```
terraform init -backend-config=backend.hcl -backend-config="key=web/terraform.tfstate"
```

ΕΞΟΔΟΣ

```
…
Successfully configured the backend "s3"! Terraform will automatically
…
Terraform has been successfully initialized!
…
```

Το `init` καταγράφει τη ρύθμιση στο `.terraform/terraform.tfstate` του κεφαλαίου 4.

TERMINAL

```
jq '.backend.config | {bucket, key, region, use_lockfile}' .terraform/terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "bucket": "tfbook-state",
  "key": "web/terraform.tfstate",
  "region": "eu-central-1",
  "use_lockfile": true
}
```

Το `use_lockfile` ήρθε από το block, τα `bucket` και `region` από το αρχείο και το `key` από τη γραμμή εντολών. Αν μια τιμή δοθεί δύο φορές, κερδίζει η τελευταία. Αν λείπει υποχρεωτική τιμή, το `init` τη ζητά. Με `-input=false` σταματά με `Missing Required Value`. Με ένα αρχείο ανά περιβάλλον, όπως `dev.hcl` και `prod.hcl`, ο ίδιος κώδικας γράφει σε διαφορετικό state. Αλλάζεις περιβάλλον με `terraform init -reconfigure -backend-config=prod.hcl`. Το κεφάλαιο 20 δείχνει έναν άλλο τρόπο, τα workspaces.

ΠΑΓΙΔΑ

Οι τιμές της partial configuration γράφονται σε απλό κείμενο στο `.terraform/terraform.tfstate` και σε κάθε αποθηκευμένο plan. Μη δίνεις έτσι access keys ή άλλα μυστικά. Τα credentials του backend έρχονται από τις μεταβλητές περιβάλλοντος ή από τα αρχεία του AWS.

Το lock στην πράξη

Χρειάζεσαι μια εντολή που κρατά το state αρκετή ώρα. Το `time_sleep` του κεφαλαίου 14 περιμένει όσο του πεις όταν δημιουργείται.

TERMINAL

```
cd ..
mkdir slow
cd slow
```

main.tf

```
terraform {
  required_providers {
    time = {
      source  = "hashicorp/time"
      version = "0.14.2"
    }
  }

  backend "s3" {
    bucket       = "tfbook-state"
    key          = "team/slow.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}

resource "time_sleep" "wait" {
  create_duration = "20s"
}
```

TERMINAL

```
terraform init
```

Αντί για δεύτερο terminal, το apply τρέχει στο παρασκήνιο όπως στο κεφάλαιο 4. Η έξοδός του πηγαίνει στο `apply.log`.

TERMINAL

```
terraform apply -auto-approve > apply.log 2>&1 &
sleep 5
aws s3 ls s3://tfbook-state/team/
```

ΕΞΟΔΟΣ

```
2026-10-08 05:28:45        769 base.tfstate
2026-10-08 05:28:54        227 slow.tfstate.tflock
```

Εμφανίστηκε το `slow.tfstate.tflock`. Το `slow.tfstate` λείπει ακόμη, γιατί το πρώτο apply δεν τελείωσε. Το lock είναι ένα μικρό JSON.

TERMINAL

```
aws s3 cp s3://tfbook-state/team/slow.tfstate.tflock - | jq .
```

ΕΞΟΔΟΣ

```
{
  "ID": "e4e56115-6f11-7aa9-f125-cdd5b7ad9e76",
  "Operation": "OperationTypeApply",
  "Info": "",
  "Who": "you@laptop",
  "Version": "1.16.4",
  "Created": "2026-10-08T02:28:54.753544Z",
  "Path": "tfbook-state/team/slow.tfstate"
}
```

Έχει μοναδικό `ID`, την εντολή στο `Operation`, ποιος την έτρεξε στο `Who`, την έκδοση και την ώρα. Τρέξε plan, όπως θα έκανε την ίδια στιγμή ένας συνάδελφος.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Error acquiring the state lock
│
│ Error message: operation error S3: PutObject, https response error
│ StatusCode: 412, RequestID:
│ Js6anJ7ThznCVL8DwJ4doV9KG3y7yQAnTJV8D5ICtTixugSDo19t, HostID:
│ 9Gjjt1m+cjU4OPvX9O9/8RuvnG41MRb/18Oux2o5H5MY7ISNTlXN+Dz9IG62/ILVxhAGI0qyPfg=,
│ api error PreconditionFailed: At least one of the pre-conditions you
│ specified did not hold
│ Lock Info:
│   ID:        e4e56115-6f11-7aa9-f125-cdd5b7ad9e76
│   Path:      tfbook-state/team/slow.tfstate
│   Operation: OperationTypeApply
│   Who:       you@laptop
│   Version:   1.16.4
│   Created:   2026-10-08 02:28:54.753544 +0000 UTC
│   Info:
│
│
│ Terraform acquires a state lock to protect the state from being written
│ by multiple users at the same time. Please resolve the issue above and try
│ again. For most commands, you can disable locking with the "-lock=false"
│ flag, but this is not recommended.
╵
```

Το plan προσπάθησε να γράψει δικό του `.tflock`. Το S3 απάντησε `StatusCode: 412` με `PreconditionFailed`, γιατί το αντικείμενο υπάρχει. Το Terraform τύπωσε τότε το lock στο `Lock Info` και σταμάτησε χωρίς να αγγίξει το state.

TERMINAL

```
wait
tail -3 apply.log
aws s3 ls s3://tfbook-state/team/
```

ΕΞΟΔΟΣ

```
time_sleep.wait: Creation complete after 20s [id=2026-10-08T02:29:15Z]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
2026-10-08 05:28:45        769 base.tfstate
2026-10-08 05:29:15        686 slow.tfstate
```

Το `wait` περιμένει την εντολή του παρασκηνίου. Το apply τελείωσε, έγραψε το `slow.tfstate` και έσβησε το `.tflock`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το Terraform κλειδώνει αυτόματα το state σε κάθε λειτουργία που μπορεί να το γράψει, όπως το apply και το destroy. Κλειδώνει επίσης στο plan, όπως είδες. Αν δεν πάρει το lock, σταματά. Δεν υποστηρίζουν όλα τα backends lock. Η τεκμηρίωση κάθε backend λέει αν κλειδώνει.

Αναμονή με -lock-timeout

Σε ένα pipeline δεν θέλεις αποτυχία επειδή κάποιος έτρεχε apply εκείνη τη στιγμή. Με `-lock-timeout` το Terraform ξαναδοκιμάζει έως το όριο που του δίνεις.

TERMINAL

```
terraform apply -auto-approve -replace=time_sleep.wait > apply.log 2>&1 &
sleep 3
terraform plan -lock-timeout=60s
wait
```

ΕΞΟΔΟΣ

```
Acquiring state lock. This may take a few moments...
time_sleep.wait: Refreshing state... [id=2026-10-08T02:29:36Z]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Η γραμμή `Acquiring state lock` εμφανίζεται όταν το lock αργεί. Το plan περίμενε το apply, πήρε το lock και βρήκε `No changes`, γιατί το νέο `time_sleep` υπήρχε ήδη.

Οι προσπάθειες αραιώνουν. Η αναμονή ανάμεσά τους διπλασιάζεται από 1 έως 16 δευτερόλεπτα και μετά μένει στα 16. Οι προσπάθειες γίνονται λοιπόν περίπου στα 1, 3, 7, 15 και 31 δευτερόλεπτα. Στη δοκιμή του βιβλίου ένα `-lock-timeout=30s` δεν έφτανε. Η προσπάθεια στα 15 δευτερόλεπτα έπεφτε λίγο πριν τελειώσει το apply και η επόμενη, στα 31, μετά το όριο. Δώσε άνεση στο όριο.

ΠΑΓΙΔΑ

Το μήνυμα του lock αναφέρει το `-lock=false`, που λέει στο Terraform να μην πάρει lock. Ένα plan με `-lock=false` απλώς κινδυνεύει να διαβάσει state που αλλάζει. Ένα apply με `-lock=false` δίπλα σε άλλο apply είναι αυτό που το lock υπάρχει για να εμποδίσει. Και τα δύο γράφουν state και το δεύτερο σβήνει ό,τι έγραψε το πρώτο. Όταν ένα lock σε εμποδίζει, βρες ποιος το κρατά. Μην το παρακάμπτεις.



Ορφανό lock και force-unlock

Ένα lock το σβήνει η εντολή που το πήρε. Με Ctrl+C το Terraform σταματά με τάξη, γράφει ό,τι πρόλαβε στο state και ελευθερώνει το lock. Αν όμως η διεργασία πεθάνει απότομα, το `.tflock` μένει. Αυτό γίνεται όταν κλείσει το laptop, κοπεί το δίκτυο ή σκοτωθεί ο runner του CI. Κανείς δεν κρατά πια το lock, αλλά όλοι το βλέπουν. Λέγεται **ορφανό lock**.

Το `kill -9` σκοτώνει μια διεργασία χωρίς να την αφήσει να καθαρίσει. Το `$!` είναι το PID της τελευταίας εντολής του παρασκηνίου.

TERMINAL

```
terraform apply -auto-approve -replace=time_sleep.wait > apply.log 2>&1 &
sleep 5
kill -9 $!
```

TERMINAL

```
tail -2 apply.log
aws s3 ls s3://tfbook-state/team/
```

ΕΞΟΔΟΣ

```
time_sleep.wait: Destruction complete after 0s
time_sleep.wait: Creating...
2026-10-08 05:28:45        769 base.tfstate
2026-10-08 05:29:36        686 slow.tfstate
2026-10-08 05:29:50        227 slow.tfstate.tflock
```

Το log σταματά στο `Creating...` και το `slow.tfstate.tflock` έμεινε στο bucket. Το `slow.tfstate` έχει ακόμη την ώρα του προηγούμενου apply.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Error acquiring the state lock
│
│ Error message: operation error S3: PutObject, https response error
│ StatusCode: 412, RequestID:
│ cxmsphxfCRgT8bWscS7x2E9njwCXlbk5icZaTmcwJwohmwTmk3lV, HostID:
│ 9Gjjt1m+cjU4OPvX9O9/8RuvnG41MRb/18Oux2o5H5MY7ISNTlXN+Dz9IG62/ILVxhAGI0qyPfg=,
│ api error PreconditionFailed: At least one of the pre-conditions you
│ specified did not hold
│ Lock Info:
│   ID:        884efedc-02b5-ca80-e2a2-48f697d40596
│   Path:      tfbook-state/team/slow.tfstate
│   Operation: OperationTypeApply
│   Who:       you@laptop
│   Version:   1.16.4
│   Created:   2026-10-08 02:29:50.624935 +0000 UTC
│   Info:
│
│
│ Terraform acquires a state lock to protect the state from being written
│ by multiple users at the same time. Please resolve the issue above and try
│ again. For most commands, you can disable locking with the "-lock=false"
│ flag, but this is not recommended.
╵
```

Το σφάλμα είναι ίδιο, αλλά δεν θα φύγει όσο κι αν περιμένεις. Πρώτα βεβαιώσου ότι η εντολή του `Who` και του `Created` δεν τρέχει πουθενά. Μετά σβήσε το lock με `terraform force-unlock` και το `ID` του μηνύματος. Στο terminal σου θα το αντέγραφες από την οθόνη. Εδώ το διαβάζουμε από το αντικείμενο του lock, για να δουλεύει η εντολή σε κάθε εκτέλεση.

TERMINAL

```
LOCK_ID=$(aws s3 cp s3://tfbook-state/team/slow.tfstate.tflock - | jq -r .ID)
terraform force-unlock -force $LOCK_ID
```

ΕΞΟΔΟΣ

```
Terraform state has been successfully unlocked!

The state has been unlocked, and Terraform commands should now be able to
obtain a new lock on the remote state.
```

Χωρίς `-force` το Terraform ζητά πρώτα `yes`. Το `force-unlock` σβήνει μόνο το lock, όχι το state ή την υποδομή.

TERMINAL

```
aws s3 ls s3://tfbook-state/team/
terraform plan
```

ΕΞΟΔΟΣ

```
2026-10-08 05:28:45        769 base.tfstate
2026-10-08 05:29:36        686 slow.tfstate
time_sleep.wait: Refreshing state... [id=2026-10-08T02:29:36Z]
…
No changes. Your infrastructure matches the configuration.
…
```

Το `.tflock` έφυγε και το plan λέει `No changes`. Το apply πρόλαβε να καταστρέψει το παλιό `time_sleep`, όπως έδειξε το log, αλλά σκοτώθηκε πριν γράψει state. Για ένα `time_sleep` αυτό δεν πειράζει. Με πραγματικά resources το state μπορεί να θυμάται κάτι που σβήστηκε ή να αγνοεί κάτι που φτιάχτηκε. Το πρώτο το βρίσκει το επόμενο plan. Το δεύτερο μόνο αν ψάξεις στο cloud. Γι' αυτό ένα apply το σταματάς με Ctrl+C.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform force-unlock LOCK_ID` σβήνει ένα ορφανό lock. Θέλει το ID που δίνει το μήνυμα `Error acquiring the state lock` και δεν αλλάζει υποδομή ή state. Χρησιμοποίησέ το μόνο όταν είσαι βέβαιος ότι καμία εντολή δεν κρατά το lock. Στο local backend το lock το κρατά το λειτουργικό σύστημα και φεύγει μόλις πεθάνει η διεργασία.

Outputs άλλου project με terraform_remote_state

Η configuration `web` θέλει το πρόθεμα του `base`. Αντί να το αντιγράψεις με το χέρι, το διαβάζεις με το data source `terraform_remote_state`. Γύρνα στο `web`, που έχει ήδη backend.

TERMINAL

```
cd ../web
```

main.tf

```
data "terraform_remote_state" "base" {
  backend = "s3"

  config = {
    bucket = "tfbook-state"
    key    = "team/base.tfstate"
    region = "eu-central-1"
  }
}

resource "terraform_data" "site" {
  input = "${data.terraform_remote_state.base.outputs.prefix}-web"
}

output "site_name" {
  value = terraform_data.site.output
}
```

Το `backend` λέει τι είδους backend θα διαβάσει και το `config` έχει τα arguments του, όπως στο block `backend` του `base`. Ως data source δέχεται μεταβλητές και locals. Τα outputs του άλλου state είναι στο attribute `outputs`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.terraform_remote_state.base: Reading...
data.terraform_remote_state.base: Read complete after 0s

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # terraform_data.site will be created
  + resource "terraform_data" "site" {
      + id     = (known after apply)
      + input  = "viable-piglet-web"
      + output = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + site_name = (known after apply)
terraform_data.site: Creating...
terraform_data.site: Creation complete after 0s [id=63e06789-925a-0e1d-5de0-42f1a8ae85cd]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

site_name = "viable-piglet-web"
```

Το data source διαβάστηκε στο plan, οπότε το `input` ήταν γνωστό από πριν. Το `site_name` είναι το πρόθεμα του `base` με `-web`. Δεν χρειάστηκε κανένας provider. Το `terraform_remote_state` και το `terraform_data` ανήκουν στον ενσωματωμένο provider `terraform`.

TERRAFORM CONSOLE

```
> data.terraform_remote_state.base.outputs
{
  "prefix" = "viable-piglet"
}
> data.terraform_remote_state.base.outputs.length
╷
│ Error: Unsupported attribute
│
│   on <console-input> line 1:
│   (source code not available)
│
│ This object does not have an attribute named "length".
╵
```

Τα `outputs` έχουν μόνο το `prefix`, το μοναδικό output του root module του `base`. Το `length` είναι argument του `random_pet` και δεν φαίνεται. Η ομάδα του `base` αποφασίζει τι δημοσιεύει με τα outputs της. Τα outputs ενός child module φαίνονται μόνο αν τα ξαναδημοσιεύσει το root module.

Για να βρει τα outputs όμως, το Terraform κατεβάζει όλο το state του `base`. Όποιος τρέχει το `web` χρειάζεται ανάγνωση σε όλο το αντικείμενο, άρα και στα μυστικά του. Γι' αυτό συχνά προτιμώνται εναλλακτικές.

- **Data sources του provider.** Το `web` βρίσκει το ίδιο το resource στο cloud, για παράδειγμα ένα VPC με φίλτρο στα tags του.
- **Κοινό σημείο δημοσίευσης.** Το `base` γράφει τιμές σε μια υπηρεσία όπως το SSM Parameter Store και το `web` τις διαβάζει με data source, με δικαιώματα ανά τιμή.
- **HCP Terraform.** Το data source `tfe_outputs` δίνει μόνο τα outputs ενός workspace.

Το `terraform_remote_state` δουλεύει με κάθε backend. Για τοπικό state γράφεις `backend = "local"` και στο `config` το `path` του αρχείου.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το `terraform_remote_state` διαβάζει μόνο τα outputs του root module ενός άλλου state. Θέλει όμως ανάγνωση σε όλο το state, μαζί με τα μυστικά του.

Άλλα backends

Όλα τα backends δουλεύουν με τον ίδιο τρόπο. Ένα block `backend` με τον τύπο ως label, τα δικά του arguments, `init` και μεταφορά με `init -migrate-state`.

Backend	Πού κρατά το state	Lock
`local`	Αρχείο στον δίσκο	Ναι, στον ίδιο υπολογιστή
`s3`	Αντικείμενο σε S3 bucket	Με `use_lockfile` ή το deprecated `dynamodb_table`
`azurerm`	Blob σε Azure Storage	Ναι, με lease του blob
`gcs`	Αντικείμενο σε Google Cloud Storage	Ναι
`consul`	Key value store του Consul	Ναι
`kubernetes`	Secret στο Kubernetes	Ναι, με Lease
`http`	Οποιοδήποτε REST API	Προαιρετικά
`pg`	Βάση PostgreSQL	Ναι

Το block cloud

Για το HCP Terraform, την υπηρεσία της HashiCorp για ομάδες, υπάρχει το block `cloud`, που μπαίνει στη θέση του `backend`.

```hcl
terraform {
  cloud {
    organization = "tfbook"

    workspaces {
      name = "base"
    }
  }
}
```

Το `organization` είναι ο οργανισμός σου και το `workspaces` το workspace της configuration. Το HCP Terraform κρατά το state με lock και ιστορικό και μπορεί να τρέχει το ίδιο τα plan και τα apply. Το token το παίρνεις με `terraform login`. Τα `cloud` και `backend` δεν συνυπάρχουν στην ίδια configuration. Αν γράψεις και τα δύο, το `init` και το `validate` σταματούν με σφάλμα.

Το block `cloud` δεν είναι backend, οπότε οι σημαίες των backends δεν ισχύουν. Με `cloud` το `init` απορρίπτει τα `-migrate-state`, `-reconfigure` και `-force-copy` με `Invalid command-line option`. Για να φέρεις ένα υπάρχον state στο HCP Terraform τρέχεις σκέτο `terraform init`, που ρωτά αν θα το μεταφέρει. Το κεφάλαιο 26 είναι αφιερωμένο στο HCP Terraform.

Σε μία σελίδα

Εντολή ή ρύθμιση	Τι κάνει
`use_lockfile = true`	Lock του S3 backend με αντικείμενο `.tflock`
`terraform init -migrate-state`	Αντιγράφει το state στο νέο backend
`terraform init -reconfigure`	Ξεχνά την παλιά ρύθμιση χωρίς αντιγραφή
`-backend-config`	Partial configuration από αρχείο ή `key=value`
`terraform state pull`	Τυπώνει το remote state
`-lock-timeout=60s`	Περιμένει το lock αντί να αποτύχει
`terraform force-unlock ID`	Σβήνει ένα ορφανό lock
`data "terraform_remote_state"`	Διαβάζει τα outputs άλλου state

Ασκήσεις

Οι ασκήσεις λύνονται σε νέους φακέλους μέσα στο `~/tfbook/ch19`, στο περιβάλλον του κεφαλαίου 17, με το bucket `tfbook-state` της αρχής. Κάθε άσκηση έχει δικό της key, για να μη γράφει η μία στο state της άλλης.

ΑΣΚΗΣΗ 19.1 Ένα δεύτερο state στο ίδιο bucket

Γράψε configuration με ένα `random_integer` με όνομα `port`, από 8000 έως 8999. Πρόσθεσε ένα output `port` με την τιμή του. Κράτα το state στο bucket `tfbook-state` με key `ex1/terraform.tfstate` και lock με `use_lockfile`. Κάνε init και apply. Μετά δείξε τι υπάρχει κάτω από το `ex1/` με `aws s3 ls s3://tfbook-state/ex1/`. Η ώρα θα διαφέρει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
2026-10-08 05:30:00        770 terraform.tfstate
```

ΑΣΚΗΣΗ 19.2 Partial configuration μόνο με σημαίες

Ξεκίνα σε νέο φάκελο με το παρακάτω `backend.tf`, χωρίς αρχείο `.hcl`. Δώσε το bucket `tfbook-state`, το key `ex2/terraform.tfstate` και το region `eu-central-1` στο `terraform init` με τρεις σημαίες `-backend-config`. Πρόσθεσε ένα `terraform_data` με όνομα `note` και `input = "partial"` και κάνε apply. Τέλος δείξε τι κατέγραψε το init με `jq '.backend.config | {bucket, key, region}' .terraform/terraform.tfstate`.

backend.tf

```
terraform {
  backend "s3" {
    use_lockfile = true
  }
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
{
  "bucket": "tfbook-state",
  "key": "ex2/terraform.tfstate",
  "region": "eu-central-1"
}
```

ΑΣΚΗΣΗ 19.3 Πίσω στο local

Γράψε configuration με ένα `random_pet` με όνομα `name` και S3 backend με key `ex3/terraform.tfstate`. Κάνε init και apply. Μετά σβήσε το block `backend` και φέρε το state πίσω σε τοπικό αρχείο, χωρίς να χαθεί το resource. Δείξε τα αρχεία με `ls` και τα resources του state με `terraform state list`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
main.tf
terraform.tfstate
random_pet.name
```

ΑΣΚΗΣΗ 19.4 Περίμενε τη σειρά σου

Γράψε configuration με ένα `time_sleep` που περιμένει 15 δευτερόλεπτα κατά τη δημιουργία, με S3 backend, key `ex4/terraform.tfstate` και lock. Κάνε init. Ξεκίνα apply στο παρασκήνιο με την έξοδο στο `apply.log`, περίμενε τρία δευτερόλεπτα και τρέξε plan που περιμένει το lock έως ένα λεπτό. Στο τέλος περίμενε με `wait` να τελειώσει το apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Acquiring state lock. This may take a few moments...
time_sleep.wait: Refreshing state... [id=2026-10-08T02:30:22Z]
…
No changes. Your infrastructure matches the configuration.
…
```

ΑΣΚΗΣΗ 19.5 Ξεκλείδωσε ένα ορφανό lock

Ο φάκελος της άσκησης ξεκινά ως αντίγραφο του `slow`. Άλλαξε μόνο το key σε `ex5/terraform.tfstate` και κάνε init. Ξεκίνα apply στο παρασκήνιο, περίμενε πέντε δευτερόλεπτα και σκότωσέ το με `kill -9`. Δείξε ότι το plan αποτυγχάνει, βρες το ID του lock και ξεκλείδωσε με `terraform force-unlock -force`. Το αναμενόμενο αποτέλεσμα είναι η έξοδος του `force-unlock`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Terraform state has been successfully unlocked!

The state has been unlocked, and Terraform commands should now be able to
obtain a new lock on the remote state.
```

ΑΣΚΗΣΗ 19.6 Το πρόθεμα του base

Σε νέο φάκελο γράψε configuration χωρίς backend, δηλαδή με local state, που διαβάζει με `terraform_remote_state` το state του `base` από το `team/base.tfstate`. Δώσε ένα output `shout` με το πρόθεμα του `base` σε κεφαλαία γράμματα. Κάνε init και apply. Το δικό σου πρόθεμα θα διαφέρει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Outputs:

shout = "VIABLE-PIGLET"
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 19.1

Μια ομάδα θέλει να ορίζει το key του S3 backend από μια input variable, ώστε ο ίδιος κώδικας να γράφει σε διαφορετικό state ανά περιβάλλον. Τι κάνει;

a. Γράφει `key = var.state_key` στο block `backend`.
b. Ορίζει ένα local με το key και το χρησιμοποιεί στο block `backend`.
c. Αφήνει το key έξω από το block και το δίνει με `terraform init -backend-config`.
d. Δίνει το key με `terraform apply -var="key=dev.tfstate"`.

ΕΡΩΤΗΣΗ 19.2

Δύο μηχανικοί τρέχουν `terraform apply` την ίδια στιγμή στην ίδια configuration, με S3 backend και `use_lockfile = true`. Τι γίνεται;

a. Τα δύο apply τρέχουν μαζί και το Terraform ενώνει τις αλλαγές στο state.
b. Όποιο πάρει πρώτο το lock συνεχίζει. Το άλλο σταματά με `Error acquiring the state lock`.
c. Το δεύτερο apply περιμένει πάντα χωρίς όριο μέχρι να ελευθερωθεί το lock.
d. Το δεύτερο apply σβήνει το lock του πρώτου και συνεχίζει.

ΕΡΩΤΗΣΗ 19.3

Ένα apply στο CI σκοτώθηκε στη μέση και κάθε νέο plan αποτυγχάνει με `Error acquiring the state lock`. Βεβαιώθηκες ότι καμία εντολή δεν τρέχει. Τι κάνεις;

a. `terraform apply -lock=false` για να ολοκληρωθεί η δουλειά.
b. `terraform init -reconfigure` για να ξαναρυθμιστεί το backend.
c. `terraform plan -lock-timeout=10m` για να περιμένει να λήξει το lock.
d. `terraform force-unlock` με το ID του lock από το μήνυμα λάθους.

ΕΡΩΤΗΣΗ 19.4

Άλλαξες το key στο block `backend "s3"` μιας configuration που έχει ήδη state. Θέλεις το υπάρχον state να βρεθεί στο νέο key. Ποια εντολή τρέχεις;

a. `terraform init -migrate-state`
b. `terraform init -reconfigure`
c. `terraform init -upgrade`
d. `terraform init` χωρίς σημαίες, που μεταφέρει μόνο του το state

ΕΡΩΤΗΣΗ 19.5

Ποιες δύο προτάσεις για το data source `terraform_remote_state` ισχύουν; Επίλεξε δύο.

a. Διαβάζει κάθε attribute κάθε resource του άλλου state.
b. Δίνει μόνο τα outputs του root module του άλλου state.
c. Δουλεύει μόνο με το S3 backend.
d. Όποιος το χρησιμοποιεί χρειάζεται δικαίωμα ανάγνωσης σε όλο το άλλο state.
e. Μπορεί να γράψει τιμές στο άλλο state.

ΕΡΩΤΗΣΗ 19.6

Σωστό ή λάθος; Για να χρησιμοποιήσεις το S3 backend πρέπει να δηλώσεις τον provider `hashicorp/aws` στο `required_providers`.

a. Σωστό
b. Λάθος



ΚΕΦΑΛΑΙΟ 20

Workspaces

Μέχρι τώρα κάθε φάκελος εργασίας είχε ένα state. Σε αυτό το κεφάλαιο η ίδια configuration αποκτά πολλά states στο ίδιο backend, ένα για κάθε workspace. Βλέπεις πού γράφεται το καθένα, τοπικά και στο S3, πώς τα ξεχωρίζεις μέσα στον κώδικα με το `terraform.workspace` και πότε είναι καλύτερη επιλογή χωριστοί φάκελοι. Στο τέλος μαθαίνεις γιατί ένα workspace του HCP Terraform είναι κάτι άλλο.

Μία configuration, πολλά states

Στο κεφάλαιο 4 είδες ότι ο φάκελος εργασίας έχει ένα state και ότι το backend αποφασίζει πού γράφεται. Στο κεφάλαιο 19 το state πήγε σε S3 bucket. Και στις δύο περιπτώσεις ίσχυε ο ίδιος κανόνας. Μία configuration, ένα backend, ένα state.

Συχνά όμως θέλεις την ίδια υποδομή δύο φορές. Ένα αντίγραφο για να δοκιμάσεις μια αλλαγή πριν την περάσεις στο κανονικό ή ένα δεύτερο περιβάλλον για έναν συνάδελφο. Αν τρέξεις ξανά apply με την ίδια configuration και το ίδιο state, το Terraform δεν φτιάχνει δεύτερο αντίγραφο. Βλέπει ότι τα resources υπάρχουν και δεν κάνει τίποτα.

Το **workspace** λύνει αυτό το πρόβλημα. Είναι ένα state με όνομα. Ένας φάκελος μπορεί να έχει πολλά workspaces στο ίδιο backend και κάθε workspace κρατά το δικό του state. Τα resources του ενός δεν τα βλέπει το άλλο. Η configuration, το backend και τα credentials είναι κοινά. Αλλάζει μόνο ποιο state διαβάζει και γράφει το Terraform.

Αυτά λέγονται και **CLI workspaces**, γιατί τα διαχειρίζεσαι με την εντολή `terraform workspace`. Το HCP Terraform έχει κάτι με το ίδιο όνομα που σημαίνει άλλο πράγμα. Η διαφορά εξηγείται στο τέλος του κεφαλαίου.

Το workspace default

Φτιάξε τον φάκελο του πρώτου παραδείγματος. Η configuration φτιάχνει ένα S3 bucket με τον AWS provider του κεφαλαίου 17. Το state μένει προς το παρόν στο local backend.

TERMINAL

```
mkdir site
cd site
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "site" {
  bucket = "tfbook-site-${terraform.workspace}"
}

output "bucket" {
  value = aws_s3_bucket.site.bucket
}
```

Το καινούργιο είναι το `terraform.workspace`. Είναι μια τιμή που δίνει το ίδιο το Terraform, όπως το `path.module` του κεφαλαίου 10. Περιέχει το όνομα του workspace όπου τρέχει η εντολή. Εδώ μπαίνει στο όνομα του bucket, οπότε κάθε workspace φτιάχνει bucket με δικό του όνομα.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...
…
Initializing provider plugins...
…
- Installed hashicorp/aws v6.67.0 (signed by HashiCorp)
…
Terraform has been successfully initialized!
…
```

Δύο εντολές λένε σε ποιο workspace βρίσκεσαι.

TERMINAL

```
terraform workspace show
terraform workspace list
```

ΕΞΟΔΟΣ

```
default
* default
```

Η `show` τυπώνει μόνο το τρέχον workspace. Η `list` τυπώνει όλα τα workspaces του backend και βάζει αστερίσκο μπροστά στο τρέχον. Υπάρχει ήδη ένα, το `default`. Κάθε configuration ξεκινά σε αυτό, ακόμη κι αν δεν έχεις ακούσει ποτέ για workspaces. Όλα τα παραδείγματα των προηγούμενων κεφαλαίων έτρεξαν στο `default`.

Δες τι δίνει το `terraform.workspace` τώρα.

TERRAFORM CONSOLE

```
> terraform.workspace
"default"
> "tfbook-site-${terraform.workspace}"
"tfbook-site-default"
```

Κάνε apply.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.site: Creation complete after 3s [id=tfbook-site-default]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
bucket = "tfbook-site-default"
```

Το bucket λέγεται `tfbook-site-default`, γιατί το apply έτρεξε στο `default`.

Νέο workspace

Η `terraform workspace new` φτιάχνει ένα workspace και σε μεταφέρει αμέσως σε αυτό.

TERMINAL

```
terraform workspace new dev
```

ΕΞΟΔΟΣ

```
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Το μήνυμα το λέει καθαρά. Το νέο workspace είναι άδειο και ένα plan δεν θα δει κανένα resource. Τρέξε το.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.site will be created
…
      + bucket                      = "tfbook-site-dev"
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
  + bucket = "tfbook-site-dev"
…
```

Η configuration δεν άλλαξε, αλλά το plan θέλει να φτιάξει bucket. Το state του `dev` δεν ξέρει τίποτα για το `tfbook-site-default`. Το `terraform.workspace` γράφει τώρα `dev`, οπότε το νέο bucket λέγεται `tfbook-site-dev`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.site: Creation complete after 0s [id=tfbook-site-dev]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
```

TERMINAL

```
aws s3 ls
```

ΕΞΟΔΟΣ

```
2026-10-08 05:29:26 tfbook-site-default
2026-10-08 05:29:33 tfbook-site-dev
```



Τώρα υπάρχουν δύο buckets από την ίδια configuration, το καθένα γραμμένο σε δικό του state. Αν προσπαθήσεις να φτιάξεις ξανά το `dev` με `terraform workspace new dev`, το Terraform αρνείται με `Workspace "dev" already exists`.

> **ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS**
> Τα ονόματα των S3 buckets είναι μοναδικά σε όλο το AWS. Στο Moto το `tfbook-site-dev` είναι ελεύθερο, στο AWS μάλλον όχι. Βάλε στο όνομα και τον αριθμό του λογαριασμού σου με το `aws_caller_identity`, όπως στο κεφάλαιο 17. Τα άδεια buckets του κεφαλαίου δεν κοστίζουν τίποτα.

Πού γράφεται το state κάθε workspace

Με το local backend το state του `default` μένει εκεί που ήταν πάντα. Τα υπόλοιπα workspaces πηγαίνουν σε υποφάκελο.

TERMINAL

```
ls
find terraform.tfstate.d -type f
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
terraform.tfstate.d
terraform.tfstate.d/dev/terraform.tfstate
```

Το `terraform.tfstate` στη ρίζα του φακέλου είναι το state του `default`. Το state του `dev` βρίσκεται στο `terraform.tfstate.d/dev/terraform.tfstate`. Κάθε νέο workspace παίρνει δικό του υποφάκελο με το όνομά του μέσα στο `terraform.tfstate.d`.

Ποιο workspace είναι επιλεγμένο το γράφει το Terraform σε ένα μικρό αρχείο μέσα στο `.terraform`.

TERMINAL

```
cat .terraform/environment
```

ΕΞΟΔΟΣ

```
dev
```

Το αρχείο ανήκει στο αντίγραφο του φακέλου που έχεις εσύ. Ο φάκελος `.terraform` δεν μπαίνει σε Git, οπότε ένας συνάδελφος που κάνει clone το repository ξεκινά στο `default`, ακόμη κι αν εσύ δουλεύεις στο `dev`. Η επιλογή του workspace είναι τοπική. Τα states είναι στο backend.

Μέσα σε κάθε workspace οι διευθύνσεις των resources είναι ίδιες.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.site
```

Το `aws_s3_bucket.site` του `dev` και το `aws_s3_bucket.site` του `default` έχουν την ίδια διεύθυνση αλλά δείχνουν σε διαφορετικά buckets, γιατί ζουν σε διαφορετικά states.

Εναλλαγή με select

Η `terraform workspace select` αλλάζει workspace.

TERMINAL

```
terraform workspace select default
terraform workspace list
terraform output bucket
```

ΕΞΟΔΟΣ

```
Switched to workspace "default".
* default
  dev

"tfbook-site-default"
```

Ο αστερίσκος πέρασε στο `default` και το output διαβάστηκε από το state του `default`. Ένα workspace που δεν υπάρχει δεν επιλέγεται.

TERMINAL

```
terraform workspace select prod
```

ΕΞΟΔΟΣ

```
Workspace "prod" doesn't exist.

You can create this workspace with the "new" subcommand
or include the "-or-create" flag with the "select" subcommand.
```

Το μήνυμα προτείνει δύο δρόμους. Τη `new` ή τη σημαία `-or-create`, που επιλέγει το workspace αν υπάρχει και το φτιάχνει αν λείπει. Είναι βολική σε scripts, όπου δεν ξέρεις από πριν αν το workspace υπάρχει.

TERMINAL

```
terraform workspace select -or-create prod
```

ΕΞΟΔΟΣ

```
Created and switched to workspace "prod"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Το `prod` είναι άδειο. Θα το χρησιμοποιήσεις λίγο παρακάτω.

Η μεταβλητή TF_WORKSPACE

Σε CI pipelines δεν θέλεις να τρέχεις `select` πριν από κάθε εντολή. Η μεταβλητή περιβάλλοντος `TF_WORKSPACE` ορίζει το workspace για όσες εντολές τη βλέπουν και έχει προτεραιότητα από το `.terraform/environment`.

TERMINAL

```
TF_WORKSPACE=dev terraform output bucket
terraform workspace show
```

ΕΞΟΔΟΣ

```
"tfbook-site-dev"
prod
```

Η πρώτη εντολή διάβασε το state του `dev`. Η δεύτερη δείχνει ότι το επιλεγμένο workspace έμεινε `prod`. Η μεταβλητή ίσχυσε μόνο για την εντολή μπροστά στην οποία γράφτηκε. Αν την κάνεις `export`, ισχύει για όλες και η `select` αρνείται να αλλάξει workspace.

TERMINAL

```
export TF_WORKSPACE=dev
terraform workspace select default
```

ΕΞΟΔΟΣ

```
The selected workspace is currently overridden using the TF_WORKSPACE
environment variable.

To select a new workspace, either update this environment variable or unset
it and then run this command again.
```

TERMINAL

```
unset TF_WORKSPACE
terraform workspace show
```

ΕΞΟΔΟΣ

```
prod
```

Μετά το `unset` γύρισες στο `prod`, που είναι γραμμένο στο `.terraform/environment`.

ΠΑΓΙΔΑ

Ένα λάθος στο όνομα δεν σταματά τίποτα. Με το local backend το `TF_WORKSPACE=prdo` φτιάχνει σιωπηλά ένα νέο, άδειο workspace και το plan θέλει να στήσει τα πάντα από την αρχή.

TERMINAL

```
TF_WORKSPACE=prdo terraform plan
```

ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
```

TERMINAL

```
terraform workspace list
terraform workspace delete prdo
```

ΕΞΟΔΟΣ

```
  default
  dev
  prdo
* prod

Deleted workspace "prdo"!
```

Το `prdo` εμφανίστηκε στη λίστα χωρίς να το ζητήσεις. Ήταν άδειο, οπότε η `delete` το έσβησε χωρίς αντίρρηση.

Διαφορές ανά workspace

Τα workspaces μοιράζονται την configuration. Όταν ένα περιβάλλον χρειάζεται κάτι λίγο διαφορετικό, η διαφορά γράφεται μέσα στον κώδικα με το `terraform.workspace`. Άλλαξε το `main.tf`.

main.tf

```
terraform {
  required_providers {
```

```
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

locals {
  instance_types = {
    dev  = "t3.micro"
    prod = "m7i.large"
  }

  instance_type = lookup(local.instance_types, terraform.workspace, "t3.small")
}

resource "aws_s3_bucket" "site" {
  bucket        = "tfbook-site-${terraform.workspace}"
  force_destroy = terraform.workspace != "prod"

  tags = {
    Environment = terraform.workspace
  }
}

output "bucket" {
  value = aws_s3_bucket.site.bucket
}

output "instance_type" {
  value = local.instance_type
}
```

Το `local.instance_types` είναι ένα map με ένα μέγεθος server για κάθε workspace. Η `lookup` του κεφαλαίου 10 διαλέγει το κλειδί που ταιριάζει με το τρέχον workspace. Αν το workspace δεν υπάρχει στο map, όπως το `default`, δίνει το τρίτο argument, το `"t3.small"`. Στο κεφάλαιο 18 μια τέτοια τιμή θα πήγαινε στο `instance_type` ενός `aws_instance`. Εδώ την τυπώνει ένα output.

Το `force_destroy` είναι το argument του κεφαλαίου 17. Με τιμή `true` το destroy αδειάζει πρώτα το bucket και μετά το σβήνει. Η έκφραση `terraform.workspace != "prod"` το ενεργοποιεί παντού εκτός από το `prod`, όπου θέλεις το bucket να είναι πιο δύσκολο να χαθεί. Το tag `Environment` γράφει πάνω στο ίδιο το bucket σε ποιο workspace ανήκει, ώστε να το βλέπεις και έξω από το Terraform.

Είσαι στο `prod`. Κάνε apply χωρίς `-auto-approve` και διάβασε την ερώτηση.

TERMINAL

```
terraform apply
```

ΕΞΟΔΟΣ

```
…
      + force_destroy               = false
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
  + instance_type = "m7i.large"
…
Do you want to perform these actions in workspace "prod"?
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
instance_type = "m7i.large"
```

Στο `prod` το `force_destroy` είναι `false` και το output διάλεξε το `m7i.large`. Η ερώτηση γράφει αυτή τη φορά `in workspace "prod"`. Το Terraform προσθέτει το όνομα σε κάθε workspace εκτός από το `default`.

ΠΑΓΙΔΑ

Το plan δεν γράφει πουθενά σε ποιο workspace τρέχει και με `-auto-approve` δεν υπάρχει ούτε η ερώτηση. Αν ξεχάσεις ότι άφησες επιλεγμένο το `prod`, το επόμενο apply θα γίνει εκεί. Τρέχε `terraform workspace show` πριν από κάθε apply και βάλε το `terraform.workspace` σε ονόματα και tags, ώστε να φαίνεται μέσα στο plan.

Το console δείχνει τις ίδιες τιμές.

TERRAFORM CONSOLE

```
> terraform.workspace
"prod"
> local.instance_type
"m7i.large"
> lookup(local.instance_types, "qa", "t3.small")
"t3.small"
```

Η αλλαγή της configuration αφορά και τα άλλα workspaces. Πέρασε στο `dev` και τρέξε plan.

TERMINAL

```
terraform workspace select dev
terraform plan
```

ΕΞΟΔΟΣ

```
Switched to workspace "dev".
…
      ~ force_destroy               = false -> true
…
      ~ tags                        = {
          + "Environment" = "dev"
…
      ~ tags_all                    = {
          + "Environment" = "dev"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
  + instance_type = "t3.micro"
…
```

Το bucket του `dev` φτιάχτηκε πριν από το `force_destroy` και τα tags, οπότε το plan θέλει να το αλλάξει επί τόπου. Το `force_destroy` γίνεται από `false` σε `true` και προστίθεται το tag `Environment` με τιμή `"dev"`. Το `tags_all` είναι τα tags του resource μαζί με τα `default_tags` του κεφαλαίου 17. Το `instance_type` είναι `"t3.micro"`, όπως ορίζει το map για το `dev`. Κάθε workspace έχει δικό του state, άρα κάθε αλλαγή της configuration θέλει ένα apply σε κάθε workspace χωριστά.

TERMINAL

```
terraform apply -auto-approve
terraform workspace select default
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
…
Switched to workspace "default".
…
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
…
```

Και τα δύο apply άλλαξαν ένα resource, το bucket, με το `force_destroy` και το tag του. Τώρα και τα τρία workspaces συμφωνούν με τη configuration. Στο Moto εξαιρείται το tag του `prod`, που φτιάχτηκε μαζί με το bucket. Ένα plan εκεί θα έδειχνε ότι λείπει, για τον λόγο που εξηγεί το πλαίσιο του κεφαλαίου 17.

ΚΡΑΤΑ ΑΥΤΟ

Ένα workspace είναι ένα ακόμη state της ίδιας configuration στο ίδιο backend. Ο κώδικας, ο provider και τα credentials είναι κοινά. Οι διαφορές ανάμεσα στα workspaces γράφονται με το `terraform.workspace` μέσα στη configuration.

Διαγραφή workspace

Η `terraform workspace delete` σβήνει ένα workspace μαζί με το state του. Έχει τρεις κανόνες ασφαλείας. Ο πρώτος αφορά workspaces που έχουν ακόμη resources.

TERMINAL

```
terraform workspace delete dev
```

ΕΞΟΔΟΣ

```
Error: Workspace is not empty

Workspace "dev" is currently tracking the following resource instances:
  - aws_s3_bucket.site

Deleting this workspace would cause Terraform to lose track of any
associated remote objects, which would then require you to delete them
manually outside of Terraform. You should destroy these objects with
Terraform before deleting the workspace.

If you want to delete this workspace anyway, and have Terraform forget
about these managed objects, use the -force option to disable this safety
check.
```

Το Terraform αρνήθηκε, γιατί το state του `dev` καταγράφει το `aws_s3_bucket.site`. Αν σβηστεί το state, το bucket θα μείνει στο AWS χωρίς κανείς να το διαχειρίζεται. Ο δεύτερος κανόνας αφορά το τρέχον workspace.

TERMINAL

```
terraform workspace select dev
terraform workspace delete dev
```

ΕΞΟΔΟΣ

```
Switched to workspace "dev".
Workspace "dev" is your active workspace.

You cannot delete the currently active workspace. Please switch
to another workspace and try again.
```

Το workspace στο οποίο βρίσκεσαι δεν σβήνεται. Ο σωστός δρόμος είναι να σβήσεις πρώτα τα resources με destroy, να βγεις από το workspace και μετά να το σβήσεις.

TERMINAL

```
terraform destroy -auto-approve
terraform workspace select default
terraform workspace delete dev
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 1 destroyed.
Switched to workspace "default".
Deleted workspace "dev"!
```

Ο τρίτος κανόνας αφορά το `default`, που δεν σβήνεται ποτέ, ούτε όταν είσαι σε άλλο workspace.

TERMINAL

```
terraform workspace select prod
terraform workspace delete default
```

ΕΞΟΔΟΣ

```
Switched to workspace "prod".
Cannot delete the default workspace
```

Η σημαία `-force`

Η σημαία `-force` παρακάμπτει τον έλεγχο για resources. Δες τι σημαίνει στην πράξη με το `prod`, που έχει ακόμη το bucket του.

TERMINAL

```
terraform workspace select default
terraform workspace delete -force prod
```

ΕΞΟΔΟΣ

```
Switched to workspace "default".
Deleted workspace "prod"!
WARNING: "prod" was non-empty.
The resources managed by the deleted workspace may still exist,
but are no longer manageable by Terraform since the state has
been deleted.
```

Το workspace σβήστηκε, αλλά η προειδοποίηση λέει ότι τα resources του μπορεί να υπάρχουν ακόμη. Υπάρχουν.

TERMINAL

```
aws s3 ls
```

ΕΞΟΔΟΣ

```
2026-10-08 05:29:26 tfbook-site-default
2026-10-08 05:29:43 tfbook-site-prod
```

Το `tfbook-site-prod` είναι ακόμη στο AWS. Το Terraform το ξέχασε μαζί με το state. Ένα νέο `prod` θα ξεκινούσε άδειο και το apply του θα αποτύγχανε, γιατί το bucket με αυτό το όνομα υπάρχει ήδη. Το ορφανό bucket σβήνεται πια μόνο με το χέρι. Η `aws s3 rb` σβήνει ένα άδειο bucket.

TERMINAL

```
aws s3 rb s3://tfbook-site-prod
```

ΕΞΟΔΟΣ

```
remove_bucket: tfbook-site-prod
```

Σβήσε και το bucket του `default`, ώστε ο φάκελος να μείνει καθαρός για τις ασκήσεις.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 1 destroyed.
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά τι γίνεται με τα resources όταν σβήνεις workspace. Το `terraform workspace delete` χωρίς `-force` αρνείται όταν το state έχει resources. Με `-force` σβήνει μόνο το state. Τα πραγματικά resources μένουν και το Terraform δεν τα διαχειρίζεται πια. Το `default` δεν σβήνεται ποτέ και το τρέχον workspace δεν σβήνεται.

Workspaces στο S3 backend

Τα workspaces δεν είναι μόνο τοπικά. Τα περισσότερα remote backends τα υποστηρίζουν, ανάμεσά τους και το S3 του κεφαλαίου 19. Ένα backend που δεν τα υποστηρίζει έχει μόνο το `default`. Φτιάξε το bucket του state με τις εντολές του κεφαλαίου 19.

TERMINAL

```
aws s3api create-bucket --bucket tfbook-state \
  --create-bucket-configuration LocationConstraint=eu-central-1
aws s3api put-bucket-versioning --bucket tfbook-state \
  --versioning-configuration Status=Enabled
```

Φτιάξε έναν νέο φάκελο. Όπως στο κεφάλαιο 17, αντιγράφεις πρώτα το lock file ενός φακέλου που έχει ήδη τον provider, ώστε το `init` να τον πάρει από το plugin cache.

TERMINAL

```
cd ..
mkdir app
cd app
cp ../site/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "app" {
  bucket = "tfbook-app-${terraform.workspace}"
}
```

Η πρώτη ιδέα για χωριστό state ανά workspace είναι να βάλεις το `terraform.workspace` στο `key` του backend.

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "${terraform.workspace}/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...

│ Error: Variables not allowed
│
│   on backend.tf line 4, in terraform:
│    4:     key          = "${terraform.workspace}/terraform.tfstate"
│
│ Variables may not be used here.
```

Είναι το σφάλμα του κεφαλαίου 19. Το block `backend` ρυθμίζεται στο `init`, πριν το Terraform υπολογίσει οτιδήποτε, οπότε δέχεται μόνο σταθερές τιμές. Δεν δέχεται ούτε το `terraform.workspace`. Δεν χρειάζεται όμως. Το backend ξεχωρίζει μόνο του τα workspaces. Γράψε ένα σταθερό `key`.

backend.tf

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "app/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
Initializing the backend...
…
Successfully configured the backend "s3"! Terraform will automatically
…
Initializing provider plugins...
…
Terraform has been successfully initialized!
…
```

Κάνε apply στο `default` και φτιάξε δύο workspaces με το δικό τους apply.

TERMINAL

```
terraform apply -auto-approve
terraform workspace new dev
terraform apply -auto-approve
terraform workspace new prod
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Created and switched to workspace "dev"!
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Created and switched to workspace "prod"!
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

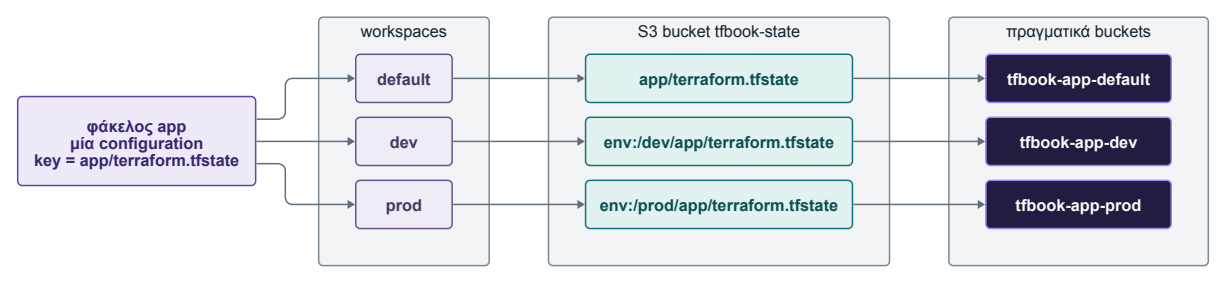
Δες τι γράφτηκε στο bucket του state.

TERMINAL

```
aws s3 ls s3://tfbook-state --recursive
```

```
ΕΞΟΔΟΣ

2026-10-08 05:30:21       2491 app/terraform.tfstate
2026-10-08 05:30:26       2467 env:/dev/app/terraform.tfstate
2026-10-08 05:30:33       2473 env:/prod/app/terraform.tfstate
```

Το state του `default` γράφτηκε ακριβώς στο `key`, στο `app/terraform.tfstate`. Τα άλλα workspaces πήγαν κάτω από το πρόθεμα `env:`, στο `env:/dev/app/terraform.tfstate` και στο `env:/prod/app/terraform.tfstate`. Η διαδρομή φτιάχνεται από το πρόθεμα, το όνομα του workspace και το `key`. Το πρόθεμα `env:` έμεινε από την εποχή που τα workspaces λέγονταν environments.



Σχήμα 20.1 · Τα workspaces μιας configuration στο S3 backend. Το default γράφει ακριβώς στο key, τα υπόλοιπα κάτω από το πρόθεμα και το όνομά τους. Κάθε state διαχειρίζεται το δικό του bucket.

Η `terraform workspace list` δεν διαβάζει τοπικό αρχείο. Ρωτά το bucket ποια states υπάρχουν με αυτό το `key`.

```
TERMINAL

terraform workspace list
```

```
ΕΞΟΔΟΣ

  default
  dev
* prod
```

Το lock του κεφαλαίου 19 είναι κι αυτό χωριστό. Όσο τρέχει ένα apply στο `dev`, το Terraform γράφει το `env:/dev/app/terraform.tfstate.tflock` και ένα plan στο `prod` δουλεύει κανονικά, γιατί κλειδώνει άλλο αρχείο.

Το workspace_key_prefix

Το πρόθεμα `env:` αλλάζει με το argument `workspace_key_prefix` του S3 backend. Άλλαξε το `backend.tf`.

backend.tf

```
terraform {
  backend "s3" {
    bucket               = "tfbook-state"
    key                  = "app/terraform.tfstate"
    workspace_key_prefix = "workspaces"
    region               = "eu-central-1"
    use_lockfile         = true
  }
}
```

Η αλλαγή στο block `backend` θέλει νέο init. Τα states υπάρχουν ήδη, οπότε χρειάζεται το `-migrate-state` του κεφαλαίου 19.

TERMINAL

```
terraform init -migrate-state
```

ΕΞΟΔΟΣ

```
…
Do you want to migrate all workspaces to "s3"?
  Both the existing "s3" backend and the newly configured "s3" backend
  support workspaces. When migrating between backends, Terraform will copy
  all workspaces (with the same names). THIS WILL OVERWRITE any conflicting
  states in the destination.

  Terraform initialization doesn't currently migrate only select workspaces.
  If you want to migrate a select number of workspaces, you must manually
  pull and push those states.

  If you answer "yes", Terraform will migrate all states. If you answer
  "no", Terraform will abort.

  Enter a value: yes


Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
…
```

Το Terraform ρωτά αν θέλεις να μεταφέρεις όλα τα workspaces. Η μεταφορά δεν γίνεται ένα ένα. Είτε αντιγράφονται όλα στη νέα θέση είτε κανένα. Ένα state που υπάρχει ήδη στη νέα θέση με το ίδιο όνομα αντικαθίσταται.

TERMINAL

```
aws s3 ls s3://tfbook-state --recursive
```

ΕΞΟΔΟΣ

```
2026-10-08 05:30:21       2491 app/terraform.tfstate
2026-10-08 05:30:26       2467 env:/dev/app/terraform.tfstate
2026-10-08 05:30:33       2473 env:/prod/app/terraform.tfstate
2026-10-08 05:30:37       2467 workspaces/dev/app/terraform.tfstate
2026-10-08 05:30:37       2473 workspaces/prod/app/terraform.tfstate
```

Τα states του `dev` και του `prod` αντιγράφηκαν στο `workspaces/dev/app/terraform.tfstate` και στο `workspaces/prod/app/terraform.tfstate`. Το `default` δεν άλλαξε θέση, γιατί το πρόθεμα ισχύει μόνο για τα υπόλοιπα workspaces. Τα παλιά αντίγραφα κάτω από το `env:` έμειναν στο bucket και δεν τα χρησιμοποιεί πια κανείς. Σβήσε τα.

TERMINAL

```
aws s3 rm s3://tfbook-state/env:/ --recursive
```

TERMINAL

```
terraform workspace list
terraform state list
```

ΕΞΟΔΟΣ

```
  default
  dev
* prod

aws_s3_bucket.app
```

Το Terraform βρίσκει τα ίδια workspaces στη νέα θέση και το `prod`, που ήταν επιλεγμένο, έχει ακόμη το bucket του.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Με το local backend το state του `default` είναι το `terraform.tfstate` και κάθε άλλο workspace γράφει στο `terraform.tfstate.d/όνομα/terraform.tfstate`. Με το S3 backend το `default` γράφει στο `key` και κάθε άλλο workspace στο `env:/όνομα/key`, όπου το `env:` αλλάζει με το `workspace_key_prefix`.

Πότε ταιριάζουν τα workspaces

Τα workspaces είναι φτιαγμένα για την ίδια configuration με μικρές διαφορές. Ταιριάζουν όταν θέλεις ένα προσωρινό αντίγραφο για να δοκιμάσεις μια αλλαγή, ένα περιβάλλον για κάθε feature branch ή



ένα προσωπικό αντίγραφο για κάθε μέλος της ομάδας. Τα αντίγραφα αυτά στήνονται και σβήνονται γρήγορα και κανείς δεν θέλει να τα κρατά σε χωριστό φάκελο.

Δεν ταιριάζουν για να χωρίσεις περιβάλλοντα που πρέπει να είναι πραγματικά απομονωμένα. Ο λόγος είναι όσα μοιράζονται.

- **Ίδιο backend.** Τα states όλων των workspaces ζουν στο ίδιο bucket και περιγράφονται από το ίδιο block `backend`. Όποιος έχει πρόσβαση στο state του `dev` έχει συνήθως πρόσβαση και στο state του `prod`.
- **Ίδια credentials.** Ο provider είναι ένας. Αν το `prod` ζει σε άλλο λογαριασμό AWS με άλλα credentials, η configuration πρέπει να τα αλλάζει με εκφράσεις, κάτι που γίνεται γρήγορα εύθραυστο.
- **Ίδιος κώδικας.** Κάθε διαφορά γίνεται συνθήκη με το `terraform.workspace`. Όσο μεγαλώνουν οι διαφορές, τόσο δυσκολότερα διαβάζεται τι ισχύει πού.
- **Αόρατη επιλογή.** Το τρέχον workspace δεν φαίνεται στον κώδικα ούτε στο plan. Με `-auto-approve` ένα apply στο λάθος workspace μοιάζει ακριβώς με ένα apply στο σωστό.

Για development, staging και production με διαφορετικούς λογαριασμούς, διαφορετικά δικαιώματα ή διαφορετικούς ανθρώπους που εγκρίνουν αλλαγές, η συνηθισμένη λύση είναι χωριστοί φάκελοι με κοινό module.

Χωριστοί φάκελοι με κοινό module

Σε αυτή τη δομή ο κοινός κώδικας ζει σε ένα module, όπως στο κεφάλαιο 15. Κάθε περιβάλλον είναι ένα μικρό root module με δικό του backend, δικό του provider και δική του κλήση του module.

TERMINAL

```
cd ..
mkdir shop
cd shop
```

modules/bucket/main.tf

```
variable "env" {
  type = string
}

resource "aws_s3_bucket" "this" {
  bucket = "tfbook-shop-${var.env}"
}

output "name" {
  value = aws_s3_bucket.this.bucket
}
```

Το module δεν ξέρει τίποτα για workspaces. Το περιβάλλον έρχεται ως input variable. Ο φάκελος του `dev` το καλεί με `env = "dev"` και κρατά το state του στο `tfbook-state`.

envs/dev/main.tf

```
terraform {
  required_providers {
```

```
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }

  backend "s3" {
    bucket       = "tfbook-state"
    key          = "shop/dev/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}

provider "aws" {
  region = "eu-central-1"
}

module "bucket" {
  source = "../../modules/bucket"
  env    = "dev"
}

output "bucket" {
  value = module.bucket.name
}
```

Ο φάκελος του `prod` είναι σχεδόν ίδιος. Κρατά όμως το state του σε άλλο bucket, το `tfbook-state-prod`. Σε πραγματική ομάδα αυτό το bucket θα ήταν σε άλλο λογαριασμό AWS, με πρόσβαση μόνο για όσους κάνουν αλλαγές στο production. Ο provider θα έπαιρνε επίσης άλλα credentials.

envs/prod/main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }

  backend "s3" {
    bucket       = "tfbook-state-prod"
    key          = "shop/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}

provider "aws" {
  region = "eu-central-1"
}

module "bucket" {
  source = "../../modules/bucket"
  env    = "prod"
}

output "bucket" {
```

```
  value = module.bucket.name
}
```

TERMINAL

```
find . -type f | sort
```

ΕΞΟΔΟΣ

```
./envs/dev/main.tf
./envs/prod/main.tf
./modules/bucket/main.tf
```

Στήσε τα δύο περιβάλλοντα. Πρώτα φτιάχνεις το bucket του state του `prod`. Μετά κάθε περιβάλλον έχει δικό του init και δικό του apply, στον δικό του φάκελο.

TERMINAL

```
aws s3api create-bucket --bucket tfbook-state-prod \
  --create-bucket-configuration LocationConstraint=eu-central-1
cd envs/dev
cp ../../../site/.terraform.lock.hcl .
terraform init > /dev/null
terraform apply -auto-approve
cd ../prod
cp ../../../site/.terraform.lock.hcl .
terraform init > /dev/null
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
bucket = "tfbook-shop-dev"
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
bucket = "tfbook-shop-prod"
```

TERMINAL

```
aws s3 ls s3://tfbook-state-prod --recursive
terraform workspace list
```

ΕΞΟΔΟΣ

```
2026-10-08 05:33:08       2596 shop/terraform.tfstate
* default
```

Το state του `prod` είναι μόνο του σε δικό του bucket. Ο φάκελος `envs/prod` έχει μόνο το workspace `default` και δεν το χρειάζεται κανένα άλλο. Το περιβάλλον το διαλέγεις με το `cd` και όχι με το `select`, οπότε φαίνεται πάντα στο path όπου βρίσκεσαι.

Η δομή έχει κόστος. Κάθε περιβάλλον έχει λίγο επαναλαμβανόμενο κώδικα, το block `terraform`, τον provider και την κλήση του module. Μια αλλαγή στο module θέλει apply σε κάθε φάκελο. Σε αντάλλαγμα κάθε περιβάλλον έχει δικό του backend, δικά του credentials και δικό του review. Η διαφορά ανάμεσα στο `dev` και στο `prod` διαβάζεται στον κώδικα.

	CLI workspaces	Χωριστοί φάκελοι
Configuration	Μία για όλα	Ένα root module ανά περιβάλλον, κοινό module
Backend και credentials	Κοινά	Χωριστά ανά περιβάλλον
Πώς διαλέγεις περιβάλλον	`select` ή `TF_WORKSPACE`	`cd` στον φάκελο
Διαφορές	Συνθήκες με `terraform.workspace`	Τιμές στην κλήση του module
Κατάλληλα για	Προσωρινά αντίγραφα, δοκιμές	Development, staging, production

Τα workspaces του HCP Terraform

Το HCP Terraform, η υπηρεσία της HashiCorp για ομάδες, οργανώνει επίσης τη δουλειά σε workspaces. Εκεί όμως η λέξη σημαίνει κάτι πολύ μεγαλύτερο. Ένα **HCP Terraform workspace** είναι μια ολόκληρη μονάδα διαχείρισης. Έχει δική του configuration, που έρχεται από ένα repository, από το CLI ή από το API, δικό του state με ιστορικό εκδόσεων, δικές του variables, ανάμεσά τους και τα credentials του cloud, δικό του ιστορικό runs και δικά του δικαιώματα ανά team. Τα workspaces ομαδοποιούνται σε projects μέσα σε ένα organization.

Ένα HCP Terraform workspace μοιάζει λοιπόν με έναν από τους χωριστούς φακέλους της προηγούμενης ενότητας, μαζί με το backend του, τα credentials του και τους ανθρώπους που μπορούν να το αλλάξουν. Δεν μοιάζει με ένα CLI workspace, που είναι μόνο ένα ακόμη state δίπλα σε άλλα. Η συνηθισμένη πρακτική στο HCP Terraform είναι ένα workspace για κάθε συνδυασμό configuration και περιβάλλοντος, για παράδειγμα `app-dev` και `app-prod`.

Τα δύο συναντιούνται στο block `cloud`. Όταν το block `workspaces` διαλέγει workspaces με `tags`, η configuration συνδέεται με όλα τα HCP Terraform workspaces που έχουν αυτά τα tags. Η `terraform workspace list` τα δείχνει και η `terraform workspace select` αλλάζει από το ένα στο άλλο. Με `name` ο φάκελος έχει ένα μόνο workspace.

```hcl
terraform {
  cloud {
    organization = "tfbook"

    workspaces {
      tags = ["app"]
    }
  }
}
```

Το κεφάλαιο 26 εξηγεί το HCP Terraform, τα projects και τα δικαιώματα αναλυτικά.

	CLI workspace	HCP Terraform workspace
Τι είναι	Ένα ακόμη state της ίδιας configuration	Μια πλήρης μονάδα διαχείρισης
State	Στο κοινό backend του φακέλου	Δικό του, με ιστορικό εκδόσεων
Variables και credentials	Κοινά για όλα τα workspaces	Δικά του για κάθε workspace
Δικαιώματα	Όσα δίνει το backend, ίδια για όλα	Ανά team, για κάθε workspace ή project
Runs	Τοπικά, στο terminal σου	Συνήθως στο HCP Terraform, με ιστορικό

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά τη διαφορά. Τα CLI workspaces είναι πολλά states μιας configuration στο ίδιο backend, με κοινά credentials. Ένα HCP Terraform workspace έχει δική του configuration, state, variables και δικαιώματα και μοιάζει περισσότερο με έναν χωριστό φάκελο εργασίας. Για απομόνωση περιβαλλόντων με διαφορετικά credentials ή δικαιώματα δεν διαλέγεις CLI workspaces.

Οι εντολές σε μία σελίδα

Εντολή	Τι κάνει
`terraform workspace show`	Τυπώνει το τρέχον workspace
`terraform workspace list`	Τυπώνει όλα τα workspaces, με `*` στο τρέχον
`terraform workspace new NAME`	Φτιάχνει workspace και περνά σε αυτό
`terraform workspace select NAME`	Περνά σε υπάρχον workspace, με `-or-create` το φτιάχνει αν λείπει
`terraform workspace delete NAME`	Σβήνει άδειο workspace, με `-force` και μη άδειο
`TF_WORKSPACE=NAME`	Ορίζει το workspace για τις εντολές που τη βλέπουν
`terraform.workspace`	Το όνομα του τρέχοντος workspace μέσα στη configuration

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch20`. Όπου η εκφώνηση λέει να ξεκινήσεις από αντίγραφο ενός φακέλου, αντέγραψε τα αρχεία `.tf` και το lock file του χωρίς το `.terraform` και τα states. Μετά τρέξε `terraform init`. Χρησιμοποίησε `-auto-approve`.

ΑΣΚΗΣΗ 20.1 Ένα workspace για δοκιμή

Ξεκίνα από αντίγραφο του `site`. Φτιάξε το workspace `test` και κάνε apply σε αυτό. Δείξε μετά τη λίστα των workspaces και το output `bucket`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  default
* test

"tfbook-site-test"
```

ΑΣΚΗΣΗ 20.2 Πού είναι τα states

Ξεκίνα από αντίγραφο του `site`. Φτιάξε το workspace `qa` και κάνε apply σε αυτό. Μετά φτιάξε το workspace `uat` χωρίς apply. Πριν τρέξεις `find terraform.tfstate.d | sort`, γράψε τι περιμένεις να δεις.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
terraform.tfstate.d
terraform.tfstate.d/qa
terraform.tfstate.d/qa/terraform.tfstate
terraform.tfstate.d/uat
```

ΑΣΚΗΣΗ 20.3 Ένα bucket μόνο στο prod

Ξεκίνα από αντίγραφο του `site`. Πρόσθεσε στο αρχείο `logs.tf` ένα `aws_s3_bucket` με όνομα `logs` που υπάρχει μόνο στο workspace `prod`, με bucket `tfbook-logs-prod`. Χρησιμοποίησε `count` όπως στο κεφάλαιο 11. Τρέξε plan στο `default`, φτιάξε το `prod` και τρέξε plan ξανά. Μην κάνεις apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
Created and switched to workspace "prod"!
…
  # aws_s3_bucket.logs[0] will be created
…
Plan: 2 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 20.4 lookup ή index

Ξεκίνα από αντίγραφο του `site` και φτιάξε το workspace `staging`. Δες στο `terraform console` τι δίνει το `local.instance_type`. Μετά άλλαξε στο `main.tf` την έκφραση του `instance_type` σε `local.instance_types[terraform.workspace]` και τρέξε plan. Τι γίνεται;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
│ Error: Invalid index
…
│ The given key does not identify an element in this collection value.
…
```

ΑΣΚΗΣΗ 20.5 Σωστή διαγραφή

Ξεκίνα από αντίγραφο του `site`. Φτιάξε το workspace `tmp`, κάνε apply και μετά σβήσε το workspace χωρίς `-force` και χωρίς να μείνει bucket στο AWS.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Destroy complete! Resources: 1 destroyed.
Switched to workspace "default".
Deleted workspace "tmp"!
```

ΑΣΚΗΣΗ 20.6 Άλλο key και άλλο πρόθεμα

Ξεκίνα από αντίγραφο του `app`. Άλλαξε στο `backend.tf` το `key` σε `web/terraform.tfstate` και το `workspace_key_prefix` σε `teams`. Τρέξε `terraform init`, φτιάξε το workspace `blue`, κάνε apply και δείξε τη λίστα των workspaces και τα αρχεία του bucket `tfbook-state` που περιέχουν `teams`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  default
* blue
…
2026-10-08 05:34:27       2473 teams/blue/web/terraform.tfstate
…
```

ΑΣΚΗΣΗ 20.7 Ένα τρίτο περιβάλλον με φάκελο

Ξεκίνα από αντίγραφο του `shop`. Πρόσθεσε το περιβάλλον `staging` ως φάκελο `envs/staging`, με το state του στο `tfbook-state` και `key` `shop/staging/terraform.tfstate`. Κάνε apply και δείξε το output `bucket`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
"tfbook-shop-staging"
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 20.1

Μια configuration χρησιμοποιεί το local backend. Τρέχεις `terraform workspace new staging` και μετά `terraform apply`. Πού γράφεται το state;

a. Στο `terraform.tfstate` στη ρίζα του φακέλου εργασίας
b. Στο `.terraform/staging.tfstate`
c. Στο `terraform.tfstate.d/staging/terraform.tfstate`
d. Στο `staging/terraform.tfstate`

ΕΡΩΤΗΣΗ 20.2

Ένα S3 backend έχει `key = "network/terraform.tfstate"` και δεν ορίζει `workspace_key_prefix`. Σε ποιο object γράφεται το state του workspace `prod`;

a. `prod/network/terraform.tfstate`
b. `env:/prod/network/terraform.tfstate`
c. `network/terraform.tfstate`
d. `network/prod/terraform.tfstate`

ΕΡΩΤΗΣΗ 20.3

Ποιες δύο προτάσεις για το workspace `default` είναι σωστές; Επίλεξε δύο.

a. Υπάρχει σε κάθε configuration, ακόμη κι αν δεν έτρεξες ποτέ εντολή `terraform workspace`.
b. Το S3 backend γράφει το state του στο `env:/default/` και το `key`.
c. Δεν μπορεί να σβηστεί με την `terraform workspace delete`.
d. Σβήνεται με `terraform workspace delete -force default`.
e. Πρέπει να το φτιάξεις με `terraform workspace new default` πριν από το πρώτο apply.

ΕΡΩΤΗΣΗ 20.4

Σωστό ή λάθος; Η εντολή `terraform workspace delete -force dev` αφήνει στο cloud τα resources που καταγράφει το state του `dev`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 20.5

Η ομάδα σου έχει development και production σε δύο διαφορετικούς λογαριασμούς AWS. Μόνο δύο άνθρωποι επιτρέπεται να αλλάζουν το production και το state του πρέπει να μην είναι προσβάσιμο από τους υπόλοιπους. Ποια οργάνωση ταιριάζει;

a. Ένας φάκελος με δύο CLI workspaces, `dev` και `prod`, στο ίδιο S3 backend
b. Ένας φάκελος με CLI workspaces και `workspace_key_prefix = "prod"`
c. Ένας φάκελος με το `terraform.workspace` στο `key` του backend
d. Χωριστοί φάκελοι ή χωριστά HCP Terraform workspaces, με χωριστό state και δικά τους credentials

ΕΡΩΤΗΣΗ 20.6

Ποια πρόταση περιγράφει σωστά τη διαφορά ανάμεσα σε ένα CLI workspace και σε ένα HCP Terraform workspace;

a. Είναι το ίδιο πράγμα. Το HCP Terraform απλώς αποθηκεύει τα CLI workspaces στο cloud.
b. Ένα CLI workspace είναι ένα ακόμη state της ίδιας configuration, ενώ ένα HCP Terraform workspace έχει δική του configuration, state, variables και δικαιώματα.
c. Ένα HCP Terraform workspace είναι ένα από πολλά states του ίδιου φακέλου, ενώ ένα CLI workspace έχει δικά του δικαιώματα ανά team.
d. Τα CLI workspaces δουλεύουν μόνο με το local backend, ενώ τα HCP Terraform workspaces μόνο με remote backend.



ΜΕΡΟΣ V

Αλλαγές σε υπάρχουσα υποδομή

Ως τώρα η υποδομή άλλαζε μόνο μέσα από το Terraform. Στην πράξη κάποιος πατά ένα κουμπί στην κονσόλα, κάποια resources υπάρχουν πριν γραφτεί ο κώδικας και τα ονόματα στον κώδικα αλλάζουν καθώς εκείνος μεγαλώνει. Σε αυτό το μέρος εντοπίζεις αλλαγές που έγιναν έξω από το Terraform, διορθώνεις το state, φέρνεις υπάρχοντα resources στη διαχείριση και αλλάζεις τη δομή του κώδικα χωρίς να σβήσεις τίποτα.



ΚΕΦΑΛΑΙΟ 21

Drift, refresh-only και εντολές state

Το Terraform δεν είναι ο μόνος που αγγίζει τον λογαριασμό σου στο AWS. Σε αυτό το κεφάλαιο αλλάζεις ένα bucket με το `aws` CLI και βλέπεις πώς το βρίσκει το plan. Μαθαίνεις να αποφασίζεις αν θα γυρίσεις μια αλλαγή πίσω ή αν θα την κρατήσεις, να ενημερώνεις το state χωρίς να αγγίξεις την υποδομή και να διορθώνεις το ίδιο το state με τις εντολές `terraform state`.

Τρεις εικόνες της ίδιας υποδομής

Σε κάθε plan συναντιούνται τρεις εικόνες. Η **configuration** λέει τι θέλεις. Το **state** λέει τι θυμάται το Terraform από το τελευταίο apply. Η πραγματική υποδομή, εδώ το AWS, είναι ό,τι υπάρχει αυτή τη στιγμή. Όσο κάθε αλλαγή περνά από το Terraform, οι τρεις συμφωνούν.

Το **drift** είναι η απόκλιση της πραγματικής υποδομής από το state, επειδή κάτι άλλαξε έξω από το Terraform. Ένας συνάδελφος άλλαξε μια ρύθμιση από την κονσόλα για να λύσει ένα πρόβλημα. Ένα script πρόσθεσε tags. Μια άλλη configuration άγγιξε το ίδιο resource. Το Terraform δεν ειδοποιείται για τίποτα από αυτά. Το ανακαλύπτει στο επόμενο refresh.

Στο κεφάλαιο 4 είδες drift σε αρχείο του provider `local`. Εκεί κάθε αλλαγή στο περιεχόμενο έμοιαζε με αρχείο που χάθηκε. Ο provider του AWS διαβάζει κάθε attribute χωριστά, οπότε το plan δείχνει ακριβώς τι άλλαξε.

Ένα bucket για να το αλλάξει κάποιος άλλος

Φτιάξε τον φάκελο του πρώτου παραδείγματος.

TERMINAL

```
mkdir drift
cd drift
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "logs" {
  bucket = "tfbook-ch21-logs"
}

resource "aws_s3_bucket_versioning" "logs" {
  bucket = aws_s3_bucket.logs.id

  versioning_configuration {
    status = "Enabled"
  }
}

output "versioning" {
  value = aws_s3_bucket_versioning.logs.versioning_configuration[0].status
}
```

Στον AWS provider οι ρυθμίσεις ενός bucket είναι χωριστά resources. Το `aws_s3_bucket_versioning` ενεργοποιεί το **versioning**, που κρατά τις παλιές εκδόσεις κάθε object, ώστε μια λάθος διαγραφή να αναιρείται. Το `status` παίρνει την τιμή `Enabled` ή `Suspended`. Το output διαβάζει το `status` από το state και θα σου δείχνει πότε άλλαξε η μνήμη του Terraform.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Outputs:

versioning = "Enabled"
```

Το `aws` CLI βλέπει το ίδιο.

TERMINAL

```
aws s3api get-bucket-versioning --bucket tfbook-ch21-logs
```

ΕΞΟΔΟΣ

```
{
    "Status": "Enabled"
}
```

Μια αλλαγή έξω από το Terraform

Η Μαρία από την ομάδα ασφάλειας ελέγχει τα buckets του λογαριασμού. Βάζει στο bucket ένα tag με τον υπεύθυνο. Σταματά επίσης το versioning, γιατί πιστεύει ότι κοστίζει χωρίς λόγο. Δουλεύει με το `aws` CLI και δεν ανοίγει τον κώδικα.

TERMINAL

```
aws s3api put-bucket-tagging --bucket tfbook-ch21-logs \
  --tagging 'TagSet=[{Key=Owner,Value=maria}]'
aws s3api put-bucket-versioning --bucket tfbook-ch21-logs \
  --versioning-configuration Status=Suspended
```

Οι δύο εντολές δεν τυπώνουν τίποτα όταν πετυχαίνουν. Τρέξε plan.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Refreshing state... [id=tfbook-ch21-logs]
aws_s3_bucket_versioning.logs: Refreshing state... [id=tfbook-ch21-logs]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_s3_bucket.logs will be updated in-place
  ~ resource "aws_s3_bucket" "logs" {
        id                          = "tfbook-ch21-logs"
      ~ tags                        = {
          - "Owner" = "maria" -> null
```

```
        }
      ~ tags_all                    = {
          - "Owner" = "maria" -> null
        }
        # (14 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

  # aws_s3_bucket_versioning.logs will be updated in-place
  ~ resource "aws_s3_bucket_versioning" "logs" {
        id                    = "tfbook-ch21-logs"
        # (3 unchanged attributes hidden)

      ~ versioning_configuration {
          ~ status     = "Suspended" -> "Enabled"
            # (1 unchanged attribute hidden)
        }
    }

Plan: 0 to add, 2 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το refresh διάβασε τα δύο αντικείμενα από το AWS και το plan προτείνει δύο αλλαγές επί τόπου, με `~`. Στο bucket θέλει να σβήσει το tag `Owner`, γιατί η configuration δεν ορίζει κανένα tag. Το `tags_all` περιέχει τα tags του resource μαζί με όσα προσθέτει το `default_tags` του κεφαλαίου 17, οπότε αλλάζει κι αυτό. Στο versioning θέλει να γυρίσει το `status` από `"Suspended"` σε `"Enabled"`.

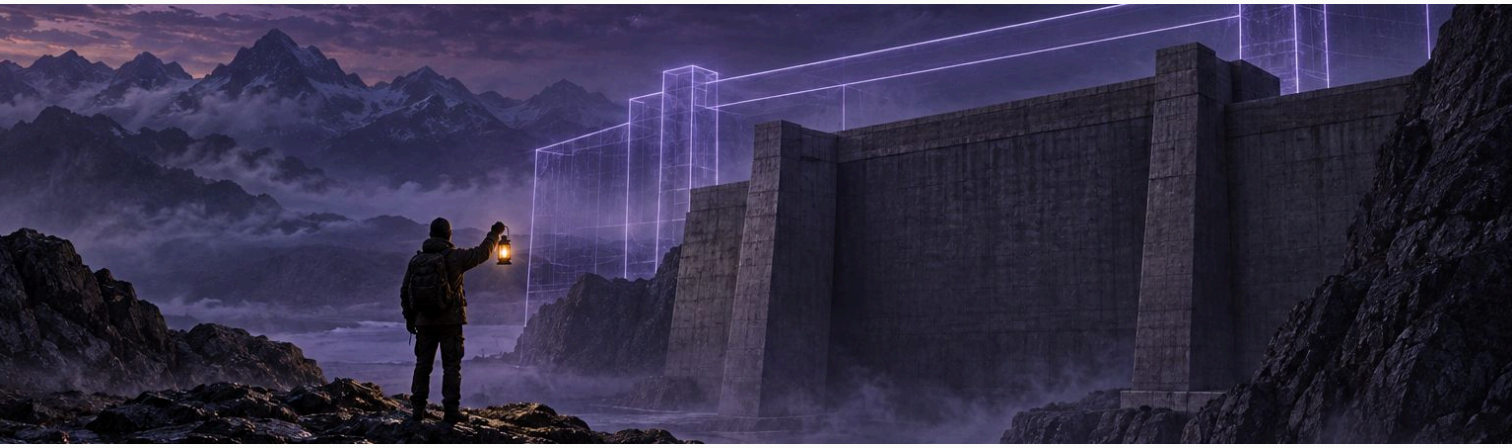
Πρόσεξε τη φορά του βέλους. Αριστερά είναι η τιμή που βρήκε το refresh στο AWS. Δεξιά είναι η τιμή της configuration. Το plan περιγράφει πώς θα επιστρέψει το AWS σε αυτό που λέει ο κώδικας. Δεν ξέρει ποιος έκανε την αλλαγή ούτε αν ήταν σκόπιμη. Για το Terraform η configuration είναι πάντα η σωστή εκδοχή.

> **ΠΑΓΙΔΑ**
> Το κανονικό plan αντιμετωπίζει κάθε drift ως λάθος που πρέπει να διορθωθεί. Ένα apply χωρίς προσεκτική ανάγνωση του plan σβήνει σιωπηλά τη δουλειά κάποιου άλλου, ακόμη κι αν εκείνη η αλλαγή έλυνε ένα πραγματικό πρόβλημα.

Πότε εμφανίζεται το «Objects have changed outside of Terraform»

Το plan από πάνω δεν είπε πουθενά ότι κάτι άλλαξε έξω από το Terraform. Έδειξε μόνο ενέργειες. Το Terraform έχει ειδική ενότητα για το drift, με τίτλο `Note: Objects have changed outside of Terraform`. Στο κανονικό plan όμως τη δείχνει μόνο όταν το drift επηρεάζει και άλλα σημεία του plan. Αυτό συμβαίνει όταν ένα αντικείμενο σβήστηκε ή όταν ένα attribute που άλλαξε το χρησιμοποιεί κάτι άλλο που αλλάζει στο ίδιο plan. Όταν το plan απλώς επαναφέρει ένα attribute, η ενότητα θα επαναλάμβανε τις ενέργειες και παραλείπεται. Ακόμη κι όταν εμφανίζεται, δείχνει μόνο τα attributes που επηρεάζουν το plan και όχι όλο το drift.

Θα τη δεις στο κανονικό plan παρακάτω, όταν σβήσεις ολόκληρο το bucket. Για να δεις όλο το drift, χωρίς ενέργειες, υπάρχει ειδικό είδος plan.



Το plan -refresh-only

ΤΕΡΜΙΝΑΛ

```
terraform plan -refresh-only
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Refreshing state... [id=tfbook-ch21-logs]
aws_s3_bucket_versioning.logs: Refreshing state... [id=tfbook-ch21-logs]

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the
last "terraform apply" which may have affected this plan:

  # aws_s3_bucket.logs has changed
  ~ resource "aws_s3_bucket" "logs" {
        id                          = "tfbook-ch21-logs"
      + tags                        = {
          + "Owner" = "maria"
        }
      ~ tags_all                    = {
          + "Owner" = "maria"
        }
        # (14 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

  # aws_s3_bucket_versioning.logs has changed
  ~ resource "aws_s3_bucket_versioning" "logs" {
        id                    = "tfbook-ch21-logs"
        # (3 unchanged attributes hidden)

      ~ versioning_configuration {
          ~ status     = "Enabled" -> "Suspended"
            # (1 unchanged attribute hidden)
        }
    }


This is a refresh-only plan, so Terraform will not take any actions to undo
```

```
these. If you were expecting these changes then you can apply this plan to
record the updated values in the Terraform state without changing any remote
objects.

──────────────────────────────────────────────────────────────────────────────

Changes to Outputs:
  ~ versioning = "Enabled" -> "Suspended"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

──────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η σημαία `-refresh-only` κάνει το refresh και σταματά εκεί. Δεν συγκρίνει με τη configuration. Συγκρίνει το state με το ανανεωμένο state, δηλαδή ό,τι θυμάται το Terraform με ό,τι βρήκε τώρα στο AWS. Η ενότητα `Objects have changed outside of Terraform` εμφανίζεται με όλες τις αλλαγές και κάθε resource γράφει `has changed`.

Τα βέλη έχουν αντίθετη φορά από το προηγούμενο plan. Το `status` πήγε από `"Enabled"` σε `"Suspended"` και το tag `Owner` εμφανίζεται με `+`, ως καινούργιο. Αριστερά είναι το state και δεξιά το AWS. Το κανονικό plan έλεγε πώς θα πάει το AWS στη configuration. Το refresh-only λέει πώς απομακρύνθηκε το AWS από το state.

Στο τέλος το Terraform λέει ότι δεν θα αναιρέσει τίποτα. Αν περίμενες τις αλλαγές, μπορείς να κάνεις apply αυτό το plan, ώστε να γραφτούν στο state χωρίς να αλλάξει κανένα αντικείμενο. Το `Changes to Outputs` δείχνει ότι μετά από ένα τέτοιο apply το output `versioning` θα έγραφε `"Suspended"`.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
>
> Το `terraform plan -refresh-only` δείχνει το drift, δηλαδή τη διαφορά ανάμεσα στο state και στην πραγματική υποδομή. Δεν προτείνει ενέργειες και δεν αλλάζει τίποτα. Είναι ο ασφαλής τρόπος να ελέγξεις τι άλλαξε έξω από το Terraform πριν αποφασίσεις τι θα κάνεις.

Δύο δρόμοι για κάθε αλλαγή

Για κάθε drift αποφασίζεις χωριστά.

1. **Επιστροφή.** Η αλλαγή ήταν λάθος. Κάνεις apply και το Terraform επαναφέρει την υποδομή σε ό,τι λέει η configuration.
2. **Αποδοχή.** Η αλλαγή ήταν σωστή. Αλλάζεις τη configuration ώστε να περιγράφει τη νέα κατάσταση. Το επόμενο plan δεν βρίσκει διαφορά.

Η ομάδα συζήτησε με τη Μαρία. Το tag με τον υπεύθυνο είναι χρήσιμο και μένει. Το versioning όμως προστατεύει τα logs από λάθος διαγραφές και πρέπει να ξαναενεργοποιηθεί. Πρόσθεσε το tag στη configuration.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
```

```
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "logs" {
  bucket = "tfbook-ch21-logs"

  tags = {
    Owner = "maria"
  }
}

resource "aws_s3_bucket_versioning" "logs" {
  bucket = aws_s3_bucket.logs.id

  versioning_configuration {
    status = "Enabled"
  }
}

output "versioning" {
  value = aws_s3_bucket_versioning.logs.versioning_configuration[0].status
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket_versioning.logs will be updated in-place
…
          ~ status     = "Suspended" -> "Enabled"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
…
```

Το bucket δεν εμφανίζεται πια στο plan, γιατί η configuration ζητά το tag που ήδη υπάρχει. Άλλαξε μόνο το versioning, που γύρισε σε `"Enabled"`. Το apply έγραψε επίσης στο state ό,τι βρήκε το refresh, άρα και το tag.

TERMINAL

```
terraform state show aws_s3_bucket.logs
```

ΕΞΟΔΟΣ

```
…
        "Owner" = "maria"
…
        "Owner" = "maria"
…
```

Το state γράφει το tag και στο `tags` και στο `tags_all`. Το tag βρίσκεται πλέον στη configuration, στο state και στο AWS. Υπάρχει και τρίτος δρόμος για attributes που αλλάζουν σκόπιμα και συνεχώς έξω από το Terraform, για παράδειγμα tags που βάζει ένα εργαλείο κοστολόγησης. Το `ignore_changes` του κεφαλαίου 14 λέει στο Terraform να μην τα επαναφέρει.

Το apply -refresh-only

Λίγες μέρες μετά η ομάδα αποφασίζει ότι τα logs δεν χρειάζονται παλιές εκδόσεις. Κάποιος σταματά ξανά το versioning από το CLI, αυτή τη φορά σκόπιμα.

TERMINAL

```
aws s3api put-bucket-versioning --bucket tfbook-ch21-logs \
  --versioning-configuration Status=Suspended
```

Θέλεις το state να καταγράψει την αλλαγή αμέσως, ώστε το output να λέει την αλήθεια σε όποιον το διαβάζει. Το `terraform apply -refresh-only` φτιάχνει το ίδιο refresh-only plan και ρωτά αν θα το γράψει στο state.

TERMINAL

```
terraform apply -refresh-only
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Refreshing state... [id=tfbook-ch21-logs]
aws_s3_bucket_versioning.logs: Refreshing state... [id=tfbook-ch21-logs]

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the
last "terraform apply" which may have affected this plan:

  # aws_s3_bucket_versioning.logs has changed
  ~ resource "aws_s3_bucket_versioning" "logs" {
        id                    = "tfbook-ch21-logs"
        # (3 unchanged attributes hidden)

      ~ versioning_configuration {
          ~ status     = "Enabled" -> "Suspended"
            # (1 unchanged attribute hidden)
        }
    }


This is a refresh-only plan, so Terraform will not take any actions to undo
these. If you were expecting these changes then you can apply this plan to
record the updated values in the Terraform state without changing any remote
objects.
```

```
─────────────────────────────────────────────────────────────────────────────

Changes to Outputs:
  ~ versioning = "Enabled" -> "Suspended"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

Would you like to update the Terraform state to reflect these detected changes?
  Terraform will write these changes to the state without modifying any real infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes


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

Outputs:

versioning = "Suspended"
```

Η ερώτηση διαφέρει από του κανονικού apply. Ρωτά αν θα ενημερωθεί το state ώστε να δείχνει τις αλλαγές που βρέθηκαν και τονίζει ότι η πραγματική υποδομή δεν θα αλλάξει. Η περίληψη γράφει `0 added, 0 changed, 0 destroyed`, γιατί το Terraform δεν άγγιξε κανένα αντικείμενο. Άλλαξε μόνο το state και το output γράφει πλέον `"Suspended"`. Τρέξε τώρα κανονικό plan.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
      ~ versioning_configuration {
          ~ status     = "Suspended" -> "Enabled"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
  ~ versioning = "Suspended" -> "Enabled"
…
```

Το plan θέλει ακόμη να ενεργοποιήσει το versioning. Το refresh-only άλλαξε το state, όχι τη configuration. Η configuration λέει ακόμη `Enabled` και το επόμενο apply θα την ακολουθήσει. Για να ολοκληρωθεί η αποδοχή, άλλαξε και τον κώδικα.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
```

```
}

resource "aws_s3_bucket" "logs" {
  bucket = "tfbook-ch21-logs"

  tags = {
    Owner = "maria"
  }
}

resource "aws_s3_bucket_versioning" "logs" {
  bucket = aws_s3_bucket.logs.id

  versioning_configuration {
    status = "Suspended"
  }
}

output "versioning" {
  value = aws_s3_bucket_versioning.logs.versioning_configuration[0].status
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Refreshing state... [id=tfbook-ch21-logs]
aws_s3_bucket_versioning.logs: Refreshing state... [id=tfbook-ch21-logs]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

ΠΑΓΙΔΑ

Το `apply -refresh-only` δεν αλλάζει τη configuration. Αν αποδεχτείς ένα drift μόνο στο state, το επόμενο κανονικό apply θα το αναιρέσει. Η αποδοχή ολοκληρώνεται όταν ο κώδικας περιγράφει τη νέα κατάσταση.

Το `apply -refresh-only` αξίζει σε δύο περιπτώσεις. Η πρώτη είναι όταν outputs ή άλλες configurations διαβάζουν το state και πρέπει να δουν τις πραγματικές τιμές πριν προλάβεις να αλλάξεις τον κώδικα. Τέτοιο παράδειγμα είναι το `terraform_remote_state` του κεφαλαίου 19. Η δεύτερη είναι όταν ένα αντικείμενο σβήστηκε σκόπιμα και θέλεις να το ξεχάσει το state. Ένα `apply -refresh-only` το αφαιρεί, όπως θα δεις σε μία από τις ασκήσεις.

Plan χωρίς refresh

Το refresh κοστίζει. Για κάθε resource ο provider κάνει κλήσεις στο API και για ένα bucket κάνει πολλές, μία για κάθε ρύθμιση. Σε configuration με χιλιάδες resources το refresh κρατά λεπτά και μπορεί να φτάσει τα όρια κλήσεων του API. Η σημαία `-refresh=false` το παραλείπει. Το plan συγκρίνει τότε τη configuration απευθείας με το state.

Άλλαξε από το CLI τον υπεύθυνο του bucket και σύγκρινε δύο plans.

TERMINAL

```
aws s3api put-bucket-tagging --bucket tfbook-ch21-logs \
  --tagging 'TagSet=[{Key=Owner,Value=nikos}]'
terraform plan -refresh=false
```

ΕΞΟΔΟΣ

```
No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Δεν υπάρχουν γραμμές `Refreshing state...` και το plan λέει ότι δεν υπάρχουν αλλαγές. Το state γράφει `maria` όπως και η configuration, οπότε δεν βρήκε διαφορά. Το AWS δεν το ρώτησε καθόλου.

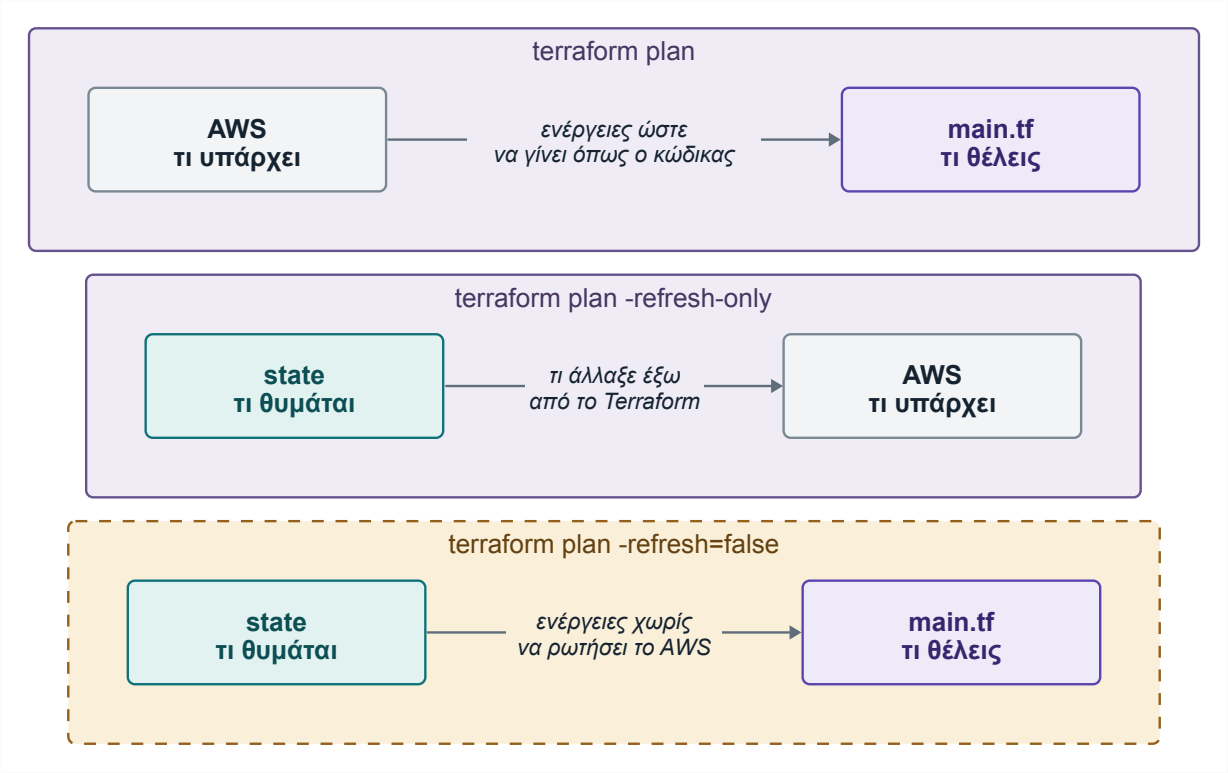
TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
          ~ "Owner" = "nikos" -> "maria"
…
          ~ "Owner" = "nikos" -> "maria"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

Με refresh το plan βρίσκει το `"nikos"` και θέλει να το γυρίσει σε `"maria"`, στο `tags` και στο `tags_all`. Το `-refresh=false` έχει νόημα όταν ξέρεις ότι τίποτα δεν άλλαξε έξω από το Terraform και θέλεις γρήγορα να δεις τι κάνει μια αλλαγή στον κώδικα, για παράδειγμα όταν τρέχεις πολλά plans στη σειρά σε μεγάλη υποδομή. Δεν το χρησιμοποιείς για το plan που θα εφαρμόσεις, γιατί κρύβει το drift.



Σχήμα 21.1 · Κάθε plan συγκρίνει δύο από τις τρεις εικόνες της υποδομής. Το βέλος έχει τη φορά του συμβόλου -> στην έξοδο. Το κανονικό plan δείχνει πώς θα επιστρέψει το AWS στη configuration, το refresh-only πώς απομακρύνθηκε το AWS από το state και το -refresh=false συγκρίνει μόνο κώδικα με μνήμη.

Το παλιό terraform refresh

Πριν από το `-refresh-only` υπήρχε η εντολή `terraform refresh`. Κάνει ό,τι και το `terraform apply -refresh-only -auto-approve`. Ενημερώνει το state χωρίς να δείξει τι άλλαξε και χωρίς να ρωτήσει. Το tag γράφει ακόμη `nikos` στο AWS.

TERMINAL

```
terraform refresh
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Refreshing state... [id=tfbook-ch21-logs]
aws_s3_bucket_versioning.logs: Refreshing state... [id=tfbook-ch21-logs]

Outputs:

versioning = "Suspended"
```

Η έξοδος δεν λέει τίποτα για το tag. Το state όμως άλλαξε.

TERMINAL

```
terraform state show aws_s3_bucket.logs
```

ΕΞΟΔΟΣ

```
…
        "Owner" = "nikos"
…
        "Owner" = "nikos"
…
```

Η εντολή είναι deprecated. Δουλεύει ακόμη, αλλά η τεκμηρίωση του Terraform προτείνει στη θέση της το `terraform apply -refresh-only`. Ο λόγος είναι ο κίνδυνος. Αν τα credentials δείχνουν κατά λάθος σε άλλο λογαριασμό, ο provider δεν βρίσκει κανένα από τα αντικείμενά σου. Το refresh τα θεωρεί σβησμένα και το `terraform refresh` τα αφαιρεί από το state χωρίς ερώτηση. Το `apply -refresh-only` θα σου έδειχνε πρώτα ότι σβήστηκαν όλα και θα απαντούσες `no`.

Η configuration λέει ακόμη `maria`. Ένα apply επαναφέρει το tag.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
          ~ "Owner" = "nikos" -> "maria"
…
          ~ "Owner" = "nikos" -> "maria"
…
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
…
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform refresh` ισοδυναμεί με `terraform apply -refresh-only -auto-approve` και είναι deprecated. Ενημερώνει μόνο το state. Δεν αλλάζει ούτε την υποδομή ούτε τη configuration. Ο σημερινός τρόπος είναι το `terraform apply -refresh-only`, που δείχνει πρώτα τις αλλαγές και ζητά έγκριση. Το `-refresh=false` κάνει το αντίθετο. Παραλείπει το refresh και δεν βλέπει το drift.

Resource που σβήστηκε έξω από το Terraform

Η χειρότερη μορφή drift είναι ένα αντικείμενο που δεν υπάρχει πια. Σβήσε το bucket με το CLI. Το `aws s3 rb` σβήνει ένα άδειο bucket.

TERMINAL

```
aws s3 rb s3://tfbook-ch21-logs
terraform plan
```

ΕΞΟΔΟΣ

```
remove_bucket: tfbook-ch21-logs
…
Note: Objects have changed outside of Terraform
…
  # aws_s3_bucket.logs has been deleted
…
  # aws_s3_bucket.logs will be created
…
  # aws_s3_bucket_versioning.logs will be created
…
Plan: 2 to add, 0 to change, 0 to destroy.
…
```

Αυτή τη φορά η ενότητα `Objects have changed outside of Terraform` εμφανίζεται και στο κανονικό plan, με `has been deleted`. Το refresh δεν βρήκε το bucket και το αφαίρεσε από το ανανεωμένο state. Μαζί του έφυγε και το versioning, γιατί δεν υπάρχει versioning σε bucket που δεν υπάρχει. Η configuration ζητά και τα δύο, οπότε το plan προτείνει `2 to add`.

Οι δύο δρόμοι ισχύουν και εδώ. Αν η διαγραφή ήταν λάθος, το apply φτιάχνει νέο bucket με το ίδιο όνομα και τις ίδιες ρυθμίσεις. Τα objects που είχε το παλιό όμως δεν επιστρέφουν, γιατί το Terraform ξέρει μόνο όσα περιγράφει ο κώδικας. Αν η διαγραφή ήταν σκόπιμη, σβήνεις τα blocks από τη configuration. Εδώ ήταν λάθος.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

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



Οι εντολές terraform state

Ως τώρα το state άλλαζε μόνο ως αποτέλεσμα ενός apply. Μερικές φορές θέλεις να αλλάξεις το ίδιο το state. Να μετονομάσεις μια διεύθυνση, να ξεχάσει το Terraform ένα resource ή να επαναφέρεις ένα αντίγραφο. Το κεφάλαιο 4 εξήγησε γιατί δεν το κάνεις με editor. Το κάνεις με τις υποεντολές του `terraform state`, που γράφουν έγκυρο state, σέβονται το lock και κρατούν backup.

Φτιάξε δεύτερο φάκελο με τρία buckets. Πάρε και το lock file του `drift`, ώστε ο νέος φάκελος να χρησιμοποιεί ακριβώς την ίδια έκδοση του provider.

TERMINAL

```
cd ..
mkdir store
cd store
cp ../drift/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "assets" {
  bucket = "tfbook-ch21-assets"
}

resource "aws_s3_bucket" "archive" {
  bucket = "tfbook-ch21-archive"
}

resource "aws_s3_bucket_versioning" "archive" {
  bucket = aws_s3_bucket.archive.id

  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket" "scratch" {
  bucket = "tfbook-ch21-scratch"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

state list και state show

Το `terraform state list` του κεφαλαίου 4 τυπώνει τις διευθύνσεις όλων των resources του state.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.archive
aws_s3_bucket.assets
aws_s3_bucket.scratch
aws_s3_bucket_versioning.archive
```

Η σειρά είναι αλφαβητική, όχι η σειρά των blocks στο αρχείο. Σε μεγάλα states θέλεις λιγότερα. Η εντολή δέχεται διευθύνσεις ως φίλτρο και τη σημαία `-id`, που κρατά τα resources με συγκεκριμένο `id`.

TERMINAL

```
terraform state list aws_s3_bucket.archive
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.archive
```

TERMINAL

```
terraform state list -id=tfbook-ch21-archive
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.archive
aws_s3_bucket_versioning.archive
```

Το φίλτρο με διεύθυνση έδωσε ένα resource. Το `-id` έδωσε δύο, γιατί το `id` του `aws_s3_bucket_versioning` είναι κι αυτό το όνομα του bucket. Το `-id` απαντά στην ερώτηση «ποιο κομμάτι του κώδικα διαχειρίζεται αυτό το αντικείμενο του AWS;». Με module, το `terraform state list module.network` θα έδειχνε όσα βρίσκονται μέσα στο module `network` του κεφαλαίου 18. Ο τύπος μόνος του όμως δεν είναι διεύθυνση.

TERMINAL

```
terraform state list aws_s3_bucket
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid address
│
│   on  line 1:
│   (source code not available)
│
│ Resource specification must include a resource type and name.
╵
```

Το `terraform state show` δείχνει ένα resource με όλα τα attributes του. Θέλει διεύθυνση που δείχνει σε ένα μόνο αντικείμενο.

TERMINAL

```
terraform state show aws_s3_bucket_versioning.archive
```

ΕΞΟΔΟΣ

```
# aws_s3_bucket_versioning.archive:
resource "aws_s3_bucket_versioning" "archive" {
    bucket                = "tfbook-ch21-archive"
    expected_bucket_owner = null
    id                    = "tfbook-ch21-archive"
    region                = "eu-central-1"

    versioning_configuration {
        mfa_delete = "Disabled"
        status     = "Enabled"
    }
}
```

Για resources με `count` ή `for_each` η διεύθυνση έχει index, για παράδειγμα `'aws_s3_bucket.b["logs"]'`. Τα μονά εισαγωγικά προστατεύουν τις αγκύλες και τα διπλά εισαγωγικά από το shell.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Για να δεις τι διαχειρίζεται το Terraform χρησιμοποιείς `terraform state list`. Για ένα resource με όλα τα attributes του χρησιμοποιείς `terraform state show` και για όλο το state `terraform show`. Το `state list` δέχεται διευθύνσεις ως φίλτρο και `-id` για να βρεις ποια διεύθυνση έχει ένα πραγματικό αντικείμενο. Καμία από αυτές τις εντολές δεν αλλάζει το state.

state mv

Το bucket `assets` θα κρατά στατικά αρχεία ενός site και η ομάδα θέλει το όνομα του resource να το λέει. Άλλαξε μόνο το label του block από `assets` σε `static`. Το όνομα του bucket στο AWS μένει ίδιο.

main.tf

```
terraform {
  required_providers {
```

```
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "static" {
  bucket = "tfbook-ch21-assets"
}

resource "aws_s3_bucket" "archive" {
  bucket = "tfbook-ch21-archive"
}

resource "aws_s3_bucket_versioning" "archive" {
  bucket = aws_s3_bucket.archive.id

  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket" "scratch" {
  bucket = "tfbook-ch21-scratch"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.assets will be destroyed
  # (because aws_s3_bucket.assets is not in configuration)
…
  # aws_s3_bucket.static will be created
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Το Terraform αναγνωρίζει τα resources μόνο από τη διεύθυνσή τους. Στο state βλέπει το `aws_s3_bucket.assets`, που λείπει από τον κώδικα. Στον κώδικα βλέπει το `aws_s3_bucket.static`, που λείπει από το state. Το plan σβήνει το πρώτο και φτιάχνει το δεύτερο, με το ίδιο όνομα bucket. Αν το bucket έχει objects, το apply αποτυγχάνει. Το destroy παίρνει `BucketNotEmpty`, γιατί το argument `force_destroy` είναι `false` από προεπιλογή. Το create παίρνει `BucketAlreadyOwnedByYou`, γιατί το όνομα είναι ακόμη πιασμένο. Με `force_destroy = true` το destroy θα έσβηνε μαζί με το bucket και όλα τα objects του. Κανένα από τα δύο δεν είναι αυτό που θέλεις.

Το `terraform state mv` μεταφέρει ένα αντικείμενο του state σε νέα διεύθυνση. Δέχεται πρώτα την παλιά και μετά τη νέα. Με `-dry-run` δείχνει τι θα έκανε χωρίς να αλλάξει τίποτα.

TERMINAL

```
terraform state mv -dry-run aws_s3_bucket.assets aws_s3_bucket.static
terraform state mv aws_s3_bucket.assets aws_s3_bucket.static
```

ΕΞΟΔΟΣ

```
Would move "aws_s3_bucket.assets" to "aws_s3_bucket.static"
Move "aws_s3_bucket.assets" to "aws_s3_bucket.static"
Successfully moved 1 object(s).
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

Το state γράφει πλέον το bucket στη νέα διεύθυνση και το plan δεν έχει τίποτα να κάνει. Κανένα αντικείμενο του AWS δεν άλλαξε.

Το ίδιο αποτέλεσμα δίνει το block `moved` του κεφαλαίου 23. Η διαφορά είναι στο πότε γίνεται η αλλαγή και ποιος τη βλέπει. Το `state mv` αλλάζει το state αμέσως και χωρίς plan. Αλλάζει μόνο το state αυτού του φακέλου και του τρέχοντος workspace. Αν ο ίδιος κώδικας τρέχει σε πολλά workspaces του κεφαλαίου 20 ή είναι module που χρησιμοποιούν άλλοι, κάθε άλλο state θα δει destroy και create μέχρι να τρέξει κάποιος το ίδιο `state mv` και εκεί. Το `moved` γράφεται στον κώδικα, φαίνεται στο plan, περνά από review και εφαρμόζεται σε κάθε state που χρησιμοποιεί τον κώδικα.

state rm

Το bucket `scratch` περνά σε άλλη ομάδα, που θα το διαχειρίζεται με δική της configuration. Το bucket πρέπει να μείνει στο AWS, αλλά το Terraform αυτού του φακέλου πρέπει να το ξεχάσει. Αν σβήσεις μόνο το block, το plan θα πρότεινε destroy. Το `terraform state rm` αφαιρεί το resource από το state χωρίς να αγγίξει το αντικείμενο.

TERMINAL

```
terraform state rm aws_s3_bucket.scratch
aws s3 ls
```

ΕΞΟΔΟΣ

```
Removed aws_s3_bucket.scratch
Successfully removed 1 resource instance(s).
2026-10-08 05:34:36 tfbook-ch21-logs
2026-10-08 05:34:43 tfbook-ch21-scratch
2026-10-08 05:34:43 tfbook-ch21-archive
2026-10-08 05:34:43 tfbook-ch21-assets
```

Το state δεν έχει πια το `scratch`, αλλά το `aws s3 ls` δείχνει ότι το bucket `tfbook-ch21-scratch` υπάρχει ακόμη. Οι ημερομηνίες στη δική σου έξοδο θα είναι άλλες. Το block όμως είναι ακόμη στον κώδικα. Δες τι κάνει ένα apply.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.scratch will be created
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
╷ Error: creating S3 Bucket (tfbook-ch21-scratch): operation error S3: CreateBucket,
https response error StatusCode: 409, RequestID:
cRoQdvDykbXlodQcmUMf8ZKtu2b8gJrPs5SmQ9FhImnV2RA6jVms, HostID:
9Gjjt1m+cjU4OPvX9O9/8RuvnG41MRb/18Oux2o5H5MY7ISNTlXN+Dz9IG62/ILVxhAGI0qyPfg=,
BucketAlreadyOwnedByYou: Your previous request to create the named bucket succeeded and
you already own it.
…
```

Το Terraform δεν θυμάται πια το bucket, οπότε θέλει να το φτιάξει. Το AWS απαντά με `BucketAlreadyOwnedByYou`, γιατί το όνομα υπάρχει ήδη στον λογαριασμό σου. Σβήσε το block, για να συμφωνεί ο κώδικας με το state.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "static" {
  bucket = "tfbook-ch21-assets"
}

resource "aws_s3_bucket" "archive" {
  bucket = "tfbook-ch21-archive"
}

resource "aws_s3_bucket_versioning" "archive" {
  bucket = aws_s3_bucket.archive.id

  versioning_configuration {
    status = "Enabled"
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

ΠΑΓΙΔΑ

Μετά το `terraform state rm` σβήσε και το block από τη configuration. Αλλιώς το επόμενο apply θα προσπαθήσει να φτιάξει ξανά το resource. Στο S3 αποτυγχάνει, γιατί το όνομα είναι πιασμένο. Σε resources χωρίς μοναδικό όνομα, όπως ένας server του EC2, το apply πετυχαίνει και φτιάχνει δεύτερο αντίγραφο.

Το block `removed` του κεφαλαίου 23 κάνει το ίδιο μέσα από τον κώδικα και το plan. Αν θέλεις κάποτε να ξαναφέρεις το bucket στη διαχείριση, το κεφάλαιο 22 δείχνει το import.

Τα backup των εντολών state

Κάθε εντολή state που αλλάζει το state κρατά πρώτα αντίγραφο της προηγούμενης μορφής του.

TERMINAL

```
ls
```

ΕΞΟΔΟΣ

```
main.tf
terraform.tfstate
terraform.tfstate.1791426894.backup
terraform.tfstate.1791426901.backup
terraform.tfstate.backup
```

Τα δύο αρχεία με αριθμό στο όνομα τα έγραψαν το `state mv` και το `state rm`. Ο αριθμός είναι η ώρα της εντολής σε δευτερόλεπτα από το 1970, οπότε στον δικό σου φάκελο θα είναι άλλος. Το Terraform δεν τα σβήνει ποτέ και δεν μπορείς να τα απενεργοποιήσεις. Με τη σημαία `-backup` ορίζεις άλλο path. Το `terraform.tfstate.backup` είναι το γνωστό backup του κεφαλαίου 4, που το έγραψε το apply που απέτυχε. Τα αρχεία αυτά τα γράφει το local backend.

state pull και state push

Το `terraform state pull` διαβάζει το state από το backend και το τυπώνει ως JSON.

TERMINAL

```
terraform state pull | jq '{serial, lineage}'
```

ΕΞΟΔΟΣ

```
{
  "serial": 9,
  "lineage": "091bf111-cd06-670c-85dd-434c4f001f37"
}
```

Με το local backend είναι σαν να διαβάζεις το `terraform.tfstate`. Με το S3 backend του κεφαλαίου 19 είναι ο τρόπος να δεις το state ή να κρατήσεις αντίγραφο χωρίς να ανοίξεις το bucket. Κράτα ένα αντίγραφο πριν από το επικίνδυνο κομμάτι.

TERMINAL

```
terraform state pull > before.tfstate
```

Το `terraform state push` κάνει το αντίθετο. Παίρνει ένα αρχείο και το γράφει ως το state του backend, στη θέση ό,τι υπήρχε. Είναι η πιο επικίνδυνη εντολή state, γιατί αντικαθιστά όλη τη μνήμη του Terraform με μία κίνηση. Γι' αυτό ελέγχει τα πεδία `serial` και `lineage` του κεφαλαίου 4. Αρνείται αρχείο με άλλο `lineage`, αρχείο με μικρότερο `serial` και αρχείο με ίδιο `serial` αλλά διαφορετικό περιεχόμενο. Δοκίμασε να γράψεις το παλαιότερο backup.

TERMINAL

```
old=$(ls terraform.tfstate.*.backup | head -1)
terraform state push "$old"
```

ΕΞΟΔΟΣ

```
Failed to write state: cannot import state with serial 5 over newer state with serial 9
```

Το backup γράφτηκε πριν από το `state mv` και έχει μικρότερο `serial`, οπότε το Terraform αρνήθηκε. Η σημαία `-force` παρακάμπτει τους ελέγχους. Δες τι προκαλεί.

TERMINAL

```
terraform state push -force "$old"
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.assets will be destroyed
  # (because aws_s3_bucket.assets is not in configuration)
…
  # aws_s3_bucket.scratch will be destroyed
  # (because aws_s3_bucket.scratch is not in configuration)
…
  # aws_s3_bucket.static will be created
…
Plan: 1 to add, 0 to change, 2 to destroy.
…
```

Το state γύρισε στην εποχή πριν από τη μετονομασία. Θυμάται ξανά το `assets` και το `scratch`, που λείπουν από τον κώδικα. Το plan θέλει να σβήσει δύο πραγματικά buckets και να φτιάξει ένα που υπάρχει ήδη. Ένα από αυτά που θα έσβηνε είναι το bucket που παραδόθηκε σε άλλη ομάδα. Ένα apply εδώ θα ήταν καταστροφή.

Επανάφερε το αντίγραφο. Το `serial` του είναι μεγαλύτερο από του state που μόλις γράφτηκε, οπότε δεν χρειάζεται `-force`.

TERMINAL

```
terraform state push before.tfstate
terraform plan
```

ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

Στο local backend ένα λάθος push διορθώνεται αν κράτησες αντίγραφο. Σε remote backend που μοιράζεται η ομάδα, ένα push με `-force` αλλάζει τη μνήμη όλων αμέσως. Το `state push` χρησιμοποιείται σπάνια, για παράδειγμα για να επαναφέρεις ένα αντίγραφο μετά από ζημιά. Πριν από αυτό κρατάς πάντα αντίγραφο του τρέχοντος state με `state pull`.

state replace-provider

Το `terraform state replace-provider` αλλάζει τη διεύθυνση του provider σε όλα τα resources του state. Το χρειάζεσαι όταν ο ίδιος provider αλλάζει source, για παράδειγμα όταν η εταιρεία σου τον σερβίρει από δικό της registry ή όταν περνάς σε fork. Δείχνει τι θα αλλάξει και ζητά `yes`. Εδώ η απάντηση ήταν `no`, οπότε το state δεν άλλαξε.

TERMINAL

```
terraform state replace-provider hashicorp/aws registry.example.com/acme/aws
```

ΕΞΟΔΟΣ

```
Terraform will perform the following actions:

  ~ Updating provider:
    - registry.terraform.io/hashicorp/aws
    + registry.example.com/acme/aws

Changing 3 resources:

  aws_s3_bucket.archive
  aws_s3_bucket.static
  aws_s3_bucket_versioning.archive

…
```

ΚΡΑΤΑ ΑΥΤΟ

Οι εντολές `terraform state mv`, `state rm`, `state push` και `state replace-provider` αλλάζουν το state χωρίς plan. Δεν αφήνουν ίχνος στον κώδικα και αφορούν μόνο το state του τρέχοντος φακέλου και workspace. Τα blocks `moved` και `removed` του κεφαλαίου 23 κάνουν τις συνηθισμένες από αυτές τις δουλειές μέσα από τον κώδικα και το plan. Προτίμησέ τα. Πριν από κάθε εντολή που αλλάζει το state κράτα αντίγραφο με `terraform state pull`.

Σε μία σελίδα

Εντολή	Τι αλλάζει	Αγγίζει την υποδομή
`terraform plan`	Τίποτα. Δείχνει πώς θα επιστρέψει η υποδομή στη configuration	Όχι
`terraform plan -refresh-only`	Τίποτα. Δείχνει το drift	Όχι
`terraform apply -refresh-only`	Το state, μετά από έγκριση	Όχι
`terraform refresh`	Το state, χωρίς έγκριση. Deprecated	Όχι
`terraform plan -refresh=false`	Τίποτα. Συγκρίνει configuration με state	Όχι
`terraform state list`, `show`, `pull`	Τίποτα	Όχι
`terraform state mv`	Τη διεύθυνση ενός resource στο state	Όχι
`terraform state rm`	Αφαιρεί resources από το state	Όχι
`terraform state push`	Αντικαθιστά όλο το state	Όχι
`terraform apply`	Την υποδομή και το state	Ναι

Ασκήσεις

Κάθε άσκηση ξεκινά από αντίγραφο του φακέλου `store` χωρίς το state του. Έτσι έχεις έτοιμο το lock file. Ξαναγράψε το `main.tf` όπως ζητά η εκφώνηση, με τα ονόματα buckets που δίνει, γιατί όλα ζουν στον ίδιο λογαριασμό. Τρέξε `terraform init` και για το apply χρησιμοποίησε `-auto-approve`.

ΑΣΚΗΣΗ 21.1 Η φορά του βέλους

Γράψε configuration με ένα bucket `tfbook-ch21-ex1` και όνομα resource `data`. Πρόσθεσε versioning με `status = "Enabled"`. Κάνε apply. Σταμάτησε το versioning με το `aws` CLI. Τρέξε πρώτα `terraform plan -refresh-only` και μετά `terraform plan`. Σύγκρινε τις γραμμές του `status` στα δύο plans.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
          ~ status     = "Enabled" -> "Suspended"
…
This is a refresh-only plan, so Terraform will not take any actions to undo
…
          ~ status     = "Suspended" -> "Enabled"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 21.2 Αποδοχή μέσα από τον κώδικα

Γράψε configuration με ένα bucket `tfbook-ch21-ex2` και όνομα resource `reports`. Πρόσθεσε versioning με `status = "Enabled"`. Κάνε apply. Σταμάτησε το versioning με το CLI. Η αλλαγή ήταν σκόπιμη. Αποδέξου την αλλάζοντας μόνο τη configuration και κάνε apply. Μετά απόδειξε με `terraform plan -refresh-only` ότι και το state ενημερώθηκε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
No changes. Your infrastructure still matches the configuration.
…
```

ΑΣΚΗΣΗ 21.3 Το bucket που σβήστηκε

Γράψε configuration με ένα bucket `tfbook-ch21-ex3` και όνομα resource `temp`. Κάνε apply. Σβήσε το bucket με `aws s3 rb`. Η διαγραφή ήταν σκόπιμη. Κάνε το state να το ξεχάσει χωρίς να το ξαναφτιάξεις και χωρίς να γράψεις τη διεύθυνση του resource. Τι δείχνει μετά το `terraform state list` και τι θα πρότεινε ένα κανονικό plan;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.temp has been deleted
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
…
Plan: 1 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 21.4 Ποια διεύθυνση έχει αυτό το bucket

Γράψε configuration με δύο buckets. Το `tfbook-ch21-ex4-a` με όνομα resource `media` και versioning ενεργό. Το `tfbook-ch21-ex4-b` με όνομα resource `backup`. Κάνε apply. Βρες με μία εντολή όλες τις διευθύνσεις του state που έχουν `id` ίσο με `tfbook-ch21-ex4-a`. Μετά τύπωσε από το state μόνο τη γραμμή του `status` του versioning.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_s3_bucket.media
aws_s3_bucket_versioning.media
…
        status     = "Enabled"
…
```

ΑΣΚΗΣΗ 21.5 Μετονομασία χωρίς αντικατάσταση

Γράψε configuration με ένα bucket `tfbook-ch21-ex5` και όνομα resource `old`. Κάνε apply. Μετονόμασε το resource σε `invoices` στον κώδικα και κάνε το state να ακολουθήσει, ώστε το επόμενο plan να μην έχει αλλαγές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Move "aws_s3_bucket.old" to "aws_s3_bucket.invoices"
Successfully moved 1 object(s).
…
No changes. Your infrastructure matches the configuration.
…
```

ΑΣΚΗΣΗ 21.6 Παράδοση σε άλλη ομάδα

Γράψε configuration με δύο buckets. Το `tfbook-ch21-ex6-keep` με όνομα resource `keep` και το `tfbook-ch21-ex6-give` με όνομα resource `give`. Κάνε apply. Το `give` περνά σε άλλη ομάδα. Κάνε το Terraform να το ξεχάσει χωρίς να σβηστεί, ώστε το επόμενο plan να μην έχει αλλαγές. Τύπωσε το state και τα ονόματα των buckets της άσκησης με `aws s3api list-buckets --query "Buckets[?contains(Name, 'ex6')].Name"`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_s3_bucket.keep
[
    "tfbook-ch21-ex6-keep",
    "tfbook-ch21-ex6-give"
]
```

ΑΣΚΗΣΗ 21.7 Η άμυνα του state push

Γράψε configuration με ένα bucket `tfbook-ch21-ex7` και όνομα resource `notes`. Κάνε apply και κράτησε αντίγραφο του state στο `v1.tfstate`. Πρόσθεσε δεύτερο bucket `tfbook-ch21-ex7-extra` με όνομα resource `extra` και κάνε apply. Δοκίμασε να γράψεις το `v1.tfstate` ως state χωρίς `-force`. Γιατί αποτυγχάνει και τι θα γινόταν με `-force`;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Failed to write state: cannot import state with serial 1 over newer state with serial 3
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 21.1

Ένας συνάδελφος άλλαξε από την κονσόλα την τιμή ενός tag σε bucket που διαχειρίζεται το Terraform. Η configuration δεν άλλαξε. Τι θα δείξει το επόμενο `terraform plan`;

a. Τίποτα, γιατί το state δεν άλλαξε.
b. Σφάλμα που ζητά να τρέξεις πρώτα `terraform refresh`.
c. Αλλαγή επί τόπου που επαναφέρει το tag στην τιμή της configuration.
d. Αντικατάσταση του bucket, γιατί άλλαξε έξω από το Terraform.

ΕΡΩΤΗΣΗ 21.2

Θέλεις να καταγράψεις στο state αλλαγές που έγιναν έξω από το Terraform, χωρίς να αλλάξει κανένα πραγματικό αντικείμενο και αφού δεις πρώτα τι θα γραφτεί. Ποια εντολή χρησιμοποιείς;

a. `terraform plan -refresh=false`
b. `terraform apply -refresh-only`
c. `terraform refresh`
d. `terraform apply -replace=aws_s3_bucket.logs`

ΕΡΩΤΗΣΗ 21.3

Εκτελείς `terraform state rm aws_s3_bucket.logs` και δεν αλλάζεις τη configuration. Ποιες δύο προτάσεις ισχύουν; Επίλεξε δύο.

a. Το bucket συνεχίζει να υπάρχει στο AWS.
b. Το bucket σβήνεται από το AWS στο επόμενο apply.
c. Η εντολή δείχνει πρώτα plan και ζητά επιβεβαίωση.
d. Το επόμενο plan προτείνει να δημιουργηθεί το `aws_s3_bucket.logs`.
e. Το Terraform σβήνει αυτόματα το block από τη configuration.

ΕΡΩΤΗΣΗ 21.4

Μετονόμασες στον κώδικα το `aws_s3_bucket.assets` σε `aws_s3_bucket.static` χωρίς να αλλάξεις το όνομα του bucket. Ποια εντολή αποτρέπει την αντικατάσταση του bucket στο επόμενο apply;

a. `terraform state rm aws_s3_bucket.assets`
b. `terraform state mv aws_s3_bucket.assets aws_s3_bucket.static`
c. `terraform apply -refresh-only`
d. `terraform state push terraform.tfstate.backup`

ΕΡΩΤΗΣΗ 21.5

Σωστό ή λάθος; Το `terraform plan -refresh=false` εντοπίζει αλλαγές που έγιναν στην υποδομή έξω από το Terraform.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 21.6

Στο AWS υπάρχει το bucket `tfbook-ch21-archive` και θέλεις να μάθεις ποια διεύθυνση του state το διαχειρίζεται. Ποια εντολή χρησιμοποιείς;

a. `terraform show tfbook-ch21-archive`
b. `terraform state show tfbook-ch21-archive`
c. `terraform state list -id=tfbook-ch21-archive`
d. `terraform state pull tfbook-ch21-archive`



ΚΕΦΑΛΑΙΟ 22

Import

Σπάνια ξεκινάς από άδειο λογαριασμό. Κάποιος έχει ήδη φτιάξει buckets από την κονσόλα, ένα script έχει γράψει parameters και ένα άλλο εργαλείο έχει αφήσει πίσω του topics. Σε αυτό το κεφάλαιο φέρνεις τέτοια resources στο state με το block `import`, αφήνεις το Terraform να γράψει μόνο του τη configuration και γνωρίζεις την παλιότερη εντολή `terraform import`. Στο τέλος ξέρεις τι αλλάζει μετά το import και ποιο λάθος δεν πρέπει να κάνεις ποτέ.

Resources που δεν έφτιαξε το Terraform

Το κεφάλαιο 4 έδειξε ότι το Terraform διαχειρίζεται μόνο ό,τι γράφει το state. Ένα αντικείμενο που υπάρχει στο AWS αλλά λείπει από το state, για το Terraform απλώς δεν υπάρχει. Αυτό ισχύει για ό,τι φτιάχτηκε από την κονσόλα, από scripts, από άλλο εργαλείο IaC όπως το CloudFormation ή από μια παλιά configuration που χάθηκε μαζί με το state της.

Φτιάξε ένα τέτοιο resource με το `aws` CLI, όπως θα το έφτιαχνε ένας συνάδελφος πριν από χρόνια. Ένα bucket με δύο tags.

TERMINAL

```
aws s3api create-bucket --bucket tfbook-ch22-legacy \
  --create-bucket-configuration LocationConstraint=eu-central-1
aws s3api put-bucket-tagging --bucket tfbook-ch22-legacy \
  --tagging 'TagSet=[{Key=team,Value=ops},{Key=owner,Value=maria}]'
```

Το `LocationConstraint` χρειάζεται σε κάθε region εκτός από το `us-east-1`. Η δεύτερη εντολή βάζει τα tags. Έλεγξε τι υπάρχει.

TERMINAL

```
aws s3api get-bucket-tagging --bucket tfbook-ch22-legacy
```

ΕΞΟΔΟΣ

```
{
    "TagSet": [
        {
            "Key": "team",
            "Value": "ops"
        },
        {
            "Key": "owner",
            "Value": "maria"
        }
    ]
}
```

Τώρα κάνε το προφανές λάθος. Γράψε ένα block για το bucket σαν να το φτιάχνεις από την αρχή. Από τα tags θυμάσαι μόνο το `team`.

TERMINAL

```
mkdir legacy
cd legacy
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "legacy" {
  bucket = "tfbook-ch22-legacy"

  tags = {
    team = "ops"
  }
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Plan: 1 to add, 0 to change, 0 to destroy.
aws_s3_bucket.legacy: Creating...
…
╷ Error: creating S3 Bucket (tfbook-ch22-legacy): operation error S3: CreateBucket,
https response error StatusCode: 409, RequestID:
xZQioNLr2rcrrYoMUWDl04KnTl5Rwu3kt2S4A9EuZVZ4LLE5ugjZ, HostID:
9Gjjt1m+cjU4OPvX9O9/8RuvnG41MRb/18Oux2o5H5MY7ISNTlXN+Dz9IG62/ILVxhAGI0qyPfg=,
BucketAlreadyOwnedByYou: Your previous request to create the named bucket succeeded and
you already own it.
…
```

Δεν υπάρχει state, οπότε το plan είπε `1 to add`. Το Terraform ζήτησε από το API να φτιάξει το bucket και το S3 απάντησε `BucketAlreadyOwnedByYou`. Το bucket υπάρχει ήδη και ανήκει στον λογαριασμό σου. Η μεγάλη γραμμή του σφάλματος περιέχει και IDs της αίτησης, που αλλάζουν σε κάθε εκτέλεση. Το χρήσιμο κομμάτι είναι στο τέλος της. Εδώ ήσουν τυχερός, γιατί τα ονόματα των buckets είναι μοναδικά και το λάθος φάνηκε αμέσως. Ένας server του EC2 δεν έχει μοναδικό όνομα. Ένα block για έναν server που ήδη υπάρχει θα έφτιαχνε δεύτερο server χωρίς κανένα σφάλμα.

Αυτό που χρειάζεσαι είναι να πεις στο Terraform ότι το αντικείμενο υπάρχει και ποια διεύθυνση της configuration του αντιστοιχεί. Αυτό είναι το **import**.

Το block import

Από την έκδοση 1.5 το import γράφεται στη configuration με ένα block `import`. Τα βασικά του arguments είναι δύο.

HCL

```
import {
  to = aws_s3_bucket.legacy
  id = "tfbook-ch22-legacy"
}
```

Το `to` είναι η διεύθυνση του resource στη configuration, ακριβώς όπως τη γράφεις σε μια αναφορά (κεφάλαιο 6). Εκεί θα «κουμπώσει» το αντικείμενο. Το `id` λέει στον provider ποιο πραγματικό αντικείμενο να βρει. Για ένα S3 bucket είναι το όνομά του.

Το block μπαίνει σε οποιοδήποτε αρχείο `.tf` του root module. Μια συνηθισμένη πρακτική είναι ένα χωριστό αρχείο `import.tf`, ώστε να βρίσκεις εύκολα τα imports και να τα σβήνεις όταν τελειώσουν.

import.tf

```
import {
  to = aws_s3_bucket.legacy
  id = "tfbook-ch22-legacy"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.legacy: Preparing import... [id=tfbook-ch22-legacy]
aws_s3_bucket.legacy: Refreshing state... [id=tfbook-ch22-legacy]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_s3_bucket.legacy will be updated in-place
  # (imported from "tfbook-ch22-legacy")
  ~ resource "aws_s3_bucket" "legacy" {
        acceleration_status         = null
        arn                         = "arn:aws:s3:::tfbook-ch22-legacy"
        bucket                      = "tfbook-ch22-legacy"
        bucket_domain_name          = "tfbook-ch22-legacy.s3.amazonaws.com"
        bucket_namespace            = "global"
        bucket_prefix               = null
        bucket_region               = "eu-central-1"
        bucket_regional_domain_name = "tfbook-ch22-legacy.s3.eu-central-1.amazonaws.com"
        force_destroy               = false
        hosted_zone_id              = "Z21DNDUVLTQW6Q"
        id                          = "tfbook-ch22-legacy"
        object_lock_enabled         = false
        policy                      = null
        region                      = "eu-central-1"
        request_payer               = null
      ~ tags                        = {
          - "owner" = "maria" -> null
            "team"  = "ops"
        }
      ~ tags_all                    = {
          - "owner" = "maria" -> null
            "team"  = "ops"
        }

        grant {
            id          = "75aa57f09aa0c8caeab4f8c24e99d10f8e7faeebf76c078efc7c6caea54ba06a"
            permissions = [
                "FULL_CONTROL",
            ]
            type        = "CanonicalUser"
            uri         = null
        }

        versioning {
            enabled    = false
            mfa_delete = false
        }
    }

Plan: 1 to import, 0 to add, 1 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Διάβασε το plan από πάνω. Η γραμμή `Preparing import... [id=tfbook-ch22-legacy]` είναι ο provider που ψάχνει το bucket με αυτό το id. Η `Refreshing state...` διαβάζει όλα τα attributes του. Η επικεφαλίδα του

resource λέει `will be updated in-place` και από κάτω `(imported from "tfbook-ch22-legacy")`. Τα attributes που ακολουθούν δεν είναι από τη configuration. Είναι όσα διάβασε ο provider από το πραγματικό bucket, για παράδειγμα το `arn` και το `region`.

Το σύμβολο `~` εμφανίζεται μόνο στα `tags` και `tags_all`. Η configuration έχει μόνο το `team`, ενώ το bucket έχει και το `owner`, οπότε το Terraform σχεδιάζει να σβήσει το `owner`. Η περίληψη τα λέει όλα μαζί. `Plan: 1 to import, 0 to add, 1 to change, 0 to destroy.` Το ίδιο apply θα κάνει import και αμέσως μετά update in place.

Μέχρι τώρα δεν έγινε τίποτα. Το plan δεν γράφει state. Το import εκτελείται στο apply, όπως κάθε άλλη ενέργεια του plan (κεφάλαιο 3). Το βλέπεις από πριν όπως βλέπεις κάθε create ή update.

> **ΠΑΓΙΔΑ**
> Μην κάνεις apply ένα import που δείχνει και `change` ή `destroy` αν δεν ξέρεις γιατί. Σχεδόν πάντα σημαίνει ότι η configuration δεν περιγράφει σωστά το αντικείμενο. Εδώ το apply θα έσβηνε το tag `owner`, που ίσως το χρειάζεται κάποιος άλλος. Διόρθωσε τη configuration μέχρι η περίληψη να λέει μόνο `to import`.

Πρόσθεσε το tag που έλειπε.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "legacy" {
  bucket = "tfbook-ch22-legacy"

  tags = {
    team  = "ops"
    owner = "maria"
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.legacy will be imported
…
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
…
```

Τώρα η επικεφαλίδα λέει απλώς `will be imported` και η περίληψη `1 to import, 0 to add, 0 to change, 0 to destroy`. Η configuration περιγράφει το bucket όπως είναι. Αυτό είναι το plan που θέλεις πριν από κάθε import.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.legacy: Importing... [id=tfbook-ch22-legacy]
aws_s3_bucket.legacy: Import complete [id=tfbook-ch22-legacy]
…
Apply complete! Resources: 1 imported, 0 added, 0 changed, 0 destroyed.
```

Το apply γράφει `Importing...` και `Import complete` αντί για `Creating...`. Η περίληψη μετρά τα imports χωριστά, `Resources: 1 imported, 0 added`.

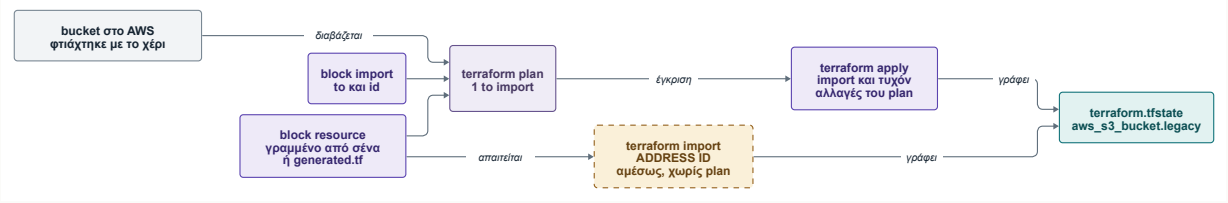
TERMINAL

```
terraform state list
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.legacy
aws_s3_bucket.legacy: Refreshing state... [id=tfbook-ch22-legacy]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το bucket βρίσκεται στο state με τη διεύθυνση `aws_s3_bucket.legacy` και το plan λέει `No changes`. Από εδώ και πέρα το Terraform το διαχειρίζεται ακριβώς όπως αν το είχε φτιάξει το ίδιο.



Σχήμα 22.1 · Το block import συνδέει ένα υπάρχον αντικείμενο με ένα block resource. Το plan δείχνει το import και το apply το γράφει στο state. Η εντολή terraform import γράφει το state αμέσως, χωρίς plan.

ΚΡΑΤΑ ΑΥΤΟ

Ένα import έχει τρία κομμάτια. Το πραγματικό αντικείμενο, ένα block `resource` που το περιγράφει και ένα block `import` που τα συνδέει. Πριν από το apply θέλεις ένα plan που λέει μόνο `to import`, χωρίς `change` και χωρίς `destroy`.

Από πού βγαίνει το id

Το `id` δεν είναι πάντα το όνομα. Κάθε resource type ορίζει τι δέχεται. Συνήθως είναι το ίδιο id που κρατά ο provider στο state για αυτόν τον τύπο. Η τεκμηρίωση ενός resource στο Terraform Registry τελειώνει συνήθως με μια ενότητα **Import**. Εκεί δίνει ένα παράδειγμα block `import` και λέει τη μορφή του `id`. Μερικά παραδείγματα από τον provider του AWS.

Resource type	Τι δίνεις ως `id`	Παράδειγμα
`aws_s3_bucket`	Το όνομα του bucket	`tfbook-ch22-legacy`
`aws_ssm_parameter`	Το όνομα του parameter	`/tfbook/ch22/db-host`
`aws_sns_topic`	Το ARN του topic	`arn:aws:sns:eu-central-1:123456789012:alerts`
`aws_instance`	Το instance ID	`i-0abc12345def67890`
`aws_vpc`	Το VPC ID	`vpc-0abc1234`
`aws_security_group`	Το ID του security group	`sg-0abc1234`
`aws_iam_role`	Το όνομα του role	`deploy`

Το `id` δεν χρειάζεται να είναι σταθερό string. Μπορεί να είναι έκφραση με variables ή locals, αρκεί η τιμή του να είναι γνωστή στο plan. Θα το δεις στην ενότητα για το `for_each`.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Η εξέταση ρωτά πού βρίσκεις τη μορφή του id. Η απάντηση είναι η τεκμηρίωση του resource στον provider, στην ενότητα Import. Το id ανήκει στο πραγματικό αντικείμενο και διαφέρει ανά resource type. Δεν είναι το όνομα που έδωσες στο block `resource`.

Configuration από το ίδιο το Terraform

Για ένα bucket με δύο arguments το block γράφτηκε εύκολα με το χέρι. Ένα πραγματικό resource όμως μπορεί να έχει δεκάδες ρυθμίσεις. Αν ξεχάσεις μία, το plan θα θέλει να την αλλάξει. Το Terraform μπορεί να γράψει το πρώτο προσχέδιο για σένα. Η σημαία `-generate-config-out` του `terraform`



`plan` βρίσκει κάθε block `import` που δεν έχει block `resource`, ζητά από τον provider όλα τα attributes του αντικειμένου και γράφει ένα block `resource` σε νέο αρχείο.

Δοκίμασέ το με ένα SSM parameter. Το Parameter Store του AWS Systems Manager κρατά ρυθμίσεις με τη μορφή όνομα και τιμή, για παράδειγμα τη διεύθυνση μιας βάσης. Στο Terraform είναι το resource `aws_ssm_parameter`. Φτιάξε ένα με το CLI.

TERMINAL

```
aws ssm put-parameter --name /tfbook/ch22/db-host --type String \
  --value db.internal.example --description "Database host" \
  --tags Key=team,Value=ops
```

ΕΞΟΔΟΣ

```
{
    "Version": 1,
    "Tier": "Standard"
}
```

Σε νέο φάκελο γράψε μόνο τον provider και το block `import`. Όπως στο κεφάλαιο 17, αντιγράφεις πρώτα το lock file ενός φακέλου που έχει κάνει ήδη init, εδώ του `legacy`, ώστε το init να πάρει τον provider από το plugin cache. Για το parameter το `id` είναι το όνομά του.

TERMINAL

```
cd ..
mkdir params
cd params
cp ../legacy/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

import {
  to = aws_ssm_parameter.db_host
  id = "/tfbook/ch22/db-host"
}
```

TERMINAL

```
terraform init
```

Ένα απλό plan δεν φτάνει, γιατί το `to` δείχνει σε block που δεν υπάρχει.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Planning failed. Terraform encountered an error while generating this plan.

╷
│ Error: Configuration for import target does not exist
│
│   on main.tf line 15, in import:
│   15:   to = aws_ssm_parameter.db_host
│
│ The configuration for the given import target aws_ssm_parameter.db_host
│ does not exist. If you wish to automatically generate config for this
│ resource, use the -generate-config-out option within terraform plan.
│ Otherwise, make sure the target resource exists within your configuration.
│ For example:
│
│   terraform plan -generate-config-out=generated.tf
╵
```

Το μήνυμα λέει τι λείπει και προτείνει την ίδια τη σημαία. Τρέξε το plan με αυτήν.

TERMINAL

```
terraform plan -generate-config-out=generated.tf
```

ΕΞΟΔΟΣ

```
aws_ssm_parameter.db_host: Preparing import... [id=/tfbook/ch22/db-host]
aws_ssm_parameter.db_host: Refreshing state... [id=/tfbook/ch22/db-host]

Planning failed. Terraform encountered an error while generating this plan.

╷
│ Warning: Config generation is experimental
│
│ Generating configuration during import is currently experimental, and the
│ generated configuration format may change in future versions.
╵
╷
│ Error: Invalid combination of arguments
│
│   with aws_ssm_parameter.db_host,
│   on generated.tf line 1:
│   (source code not available)
│
│ "insecure_value": one of `insecure_value,value,value_wo` must be specified
╵
╷
│ Error: Invalid combination of arguments
│
│   with aws_ssm_parameter.db_host,
│   on generated.tf line 16:
│   (source code not available)
│
│ "value": one of `insecure_value,value,value_wo` must be specified
```

```
╷
│
│ Error: Invalid combination of arguments
│
│   with aws_ssm_parameter.db_host,
│   on generated.tf line 17:
│   (source code not available)
│
│ "value_wo": one of `insecure_value,value,value_wo` must be specified
│
```

Ο provider βρήκε το parameter, όπως δείχνουν οι γραμμές `Preparing import` και `Refreshing state`. Η προειδοποίηση `Config generation is experimental` λέει ότι η μορφή του αρχείου μπορεί να αλλάξει σε επόμενες εκδόσεις. Μετά ακολουθούν τρία σφάλματα για το `generated.tf`. Το καθένα λέει ότι πρέπει να οριστεί ένα από τα `insecure_value`, `value` και `value_wo`. Το plan απέτυχε, αλλά το αρχείο γράφτηκε.

TERMINAL

```
cat generated.tf
```

ΕΞΟΔΟΣ

```
# __generated__ by Terraform
# Please review these resources and move them into your main configuration files.

# __generated__ by Terraform
resource "aws_ssm_parameter" "db_host" {
  allowed_pattern = null
  arn             = "arn:aws:ssm:eu-central-1:123456789012:parameter/tfbook/ch22/db-host"
  data_type       = "text"
  description     = "Database host"
  name            = "/tfbook/ch22/db-host"
  overwrite       = null
  region          = "eu-central-1"
  tags = {
    team = "ops"
  }
  tags_all = {
    team = "ops"
  }
  tier             = "Standard"
  type             = "String"
  value            = null # sensitive
  value_wo         = null # sensitive
  value_wo_version = null
}
```

Οι δύο πρώτες γραμμές σημαδεύουν το αρχείο ως `__generated__` και σου ζητούν να ελέγξεις τα resources και να τα μεταφέρεις στα κανονικά σου αρχεία. Από κάτω είναι ένα block `resource` με τη διεύθυνση του `to`. Το Terraform έγραψε τα arguments του τύπου με τις τιμές που διάβασε, ακόμη και όσα δεν έχουν τιμή και γράφονται `null`.

Το plan χάλασε επειδή το `value` γράφει `null # sensitive`. Η τιμή ενός parameter είναι sensitive (κεφάλαιο 4) και το Terraform δεν γράφει ποτέ sensitive τιμές σε παραγόμενο αρχείο. Χωρίς `value` ο provider δεν δέχεται το block. Αυτή την τιμή πρέπει να τη συμπληρώσεις εσύ.

Πρόσεξε και τα `region` και `tags_all`. Είναι τιμές που υπολογίζει ο ίδιος ο provider. Το `region` προκύπτει από τον provider και το `tags_all` από τα `tags` μαζί με τα `default_tags` (κεφάλαιο 17). Εδώ δεν προκαλούν

σφάλμα, αλλά δεν είναι ρυθμίσεις που διαλέγεις, οπότε τα σβήνεις. Αν ένα τέτοιο computed attribute προκαλέσει σφάλμα στο plan, το σβήνεις επίσης.

Το αρχείο του `-generate-config-out` πρέπει να είναι καινούργιο. Αν ξανατρέξεις την ίδια εντολή, το Terraform αρνείται.

TERMINAL

```
terraform plan -generate-config-out=generated.tf
```

ΕΞΟΔΟΣ

```
╷
│ Error: Target generated file already exists
│
│ Terraform can only write generated config into a new file. Either choose a
│ different target location or move all existing configuration out of the
│ target file, delete it and try again.
╵
```

Δεν θα γράψει ποτέ πάνω σε υπάρχον αρχείο, ώστε να μη χαθούν όσα διόρθωσες. Για νέο προσχέδιο σβήνεις το αρχείο ή δίνεις άλλο όνομα.

Βρες την τιμή που λείπει με το CLI.

TERMINAL

```
aws ssm get-parameter --name /tfbook/ch22/db-host \
  --query Parameter.Value --output text
```

ΕΞΟΔΟΣ

```
db.internal.example
```

Τώρα μετέφερε το block στο `main.tf`. Κράτησε μόνο όσα έχουν νόημα για σένα και συμπλήρωσε το `value`. Τα `data_type` και `tier` είχαν τις προεπιλεγμένες τιμές τους, οπότε φεύγουν κι αυτά.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

import {
  to = aws_ssm_parameter.db_host
  id = "/tfbook/ch22/db-host"
}
```

```
resource "aws_ssm_parameter" "db_host" {
  name        = "/tfbook/ch22/db-host"
  type        = "String"
  value       = "db.internal.example"
  description = "Database host"

  tags = {
    team = "ops"
  }
}
```

TERMINAL

```
rm generated.tf
terraform plan
```

ΕΞΟΔΟΣ

```
aws_ssm_parameter.db_host: Preparing import... [id=/tfbook/ch22/db-host]
aws_ssm_parameter.db_host: Refreshing state... [id=/tfbook/ch22/db-host]

Terraform will perform the following actions:

  # aws_ssm_parameter.db_host will be imported
    resource "aws_ssm_parameter" "db_host" {
        allowed_pattern = null
        arn             = "arn:aws:ssm:eu-central-1:123456789012:parameter/tfbook/ch22/db-host"
        data_type       = "text"
        description     = "Database host"
        has_value_wo    = false
        id              = "/tfbook/ch22/db-host"
        key_id          = null
        name            = "/tfbook/ch22/db-host"
        region          = "eu-central-1"
        tags            = {
            "team" = "ops"
        }
        tags_all        = {
            "team" = "ops"
        }
        tier            = "Standard"
        type            = "String"
        value           = (sensitive value)
        value_wo        = (write-only attribute)
        version         = 1
    }

Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το plan λέει `will be imported` και `1 to import, 0 to add, 0 to change`. Το `value` εμφανίζεται ως `(sensitive value)`, αλλά δεν έχει `~`. Η τιμή που έγραψες είναι ίδια με την πραγματική. Τα arguments που έσβησες δεν φέρνουν αλλαγή, γιατί ο provider κρατά τις τιμές που ήδη έχουν. Κάνε apply και ένα τελευταίο plan.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 imported, 0 added, 0 changed, 0 destroyed.
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_ssm_parameter.db_host: Refreshing state... [id=/tfbook/ch22/db-host]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Αυτή είναι όλη η ροή. Block `import`, προσχέδιο με `-generate-config-out`, διόρθωση με το χέρι μέχρι το plan να λέει μόνο import, apply και ένα plan χωρίς αλλαγές.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `-generate-config-out` είναι σημαία του `terraform plan`, όχι της εντολής `terraform import`. Γράφει blocks μόνο για τα `import` που δεν έχουν ήδη block `resource` και μόνο σε αρχείο που δεν υπάρχει. Είναι ακόμη experimental. Το αποτέλεσμα είναι προσχέδιο που ελέγχεις πριν μπει σε Git.

Η εντολή terraform import

Πριν από την 1.5 ο μόνος τρόπος ήταν η εντολή `terraform import ADDRESS ID`. Υπάρχει ακόμη και η εξέταση τη ρωτά. Δουλεύει διαφορετικά από το block σε τρία σημεία. Το block `resource` πρέπει να είναι ήδη γραμμένο. Η εντολή εκτελείται αμέσως, διαβάζει το αντικείμενο και γράφει το state χωρίς plan και χωρίς ερώτηση. Κάνει import ένα resource σε κάθε εκτέλεση.

Δοκίμασέ την με ένα SNS topic. Το Simple Notification Service στέλνει μηνύματα σε όσους έχουν γραφτεί σε ένα topic, για παράδειγμα σε email ή σε ουρές. Στο Terraform είναι το resource `aws_sns_topic`.

TERMINAL

```
aws sns create-topic --name tfbook-ch22-alerts --tags Key=team,Value=ops
```

ΕΞΟΔΟΣ

```
{
    "TopicArn": "arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts"
}
```

Το CLI τύπωσε το ARN του topic. Σε νέο φάκελο γράψε μόνο τον provider και δοκίμασε το import πριν γράψεις το block.

TERMINAL

```
cd ..
mkdir alerts
cd alerts
cp ../legacy/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform import aws_sns_topic.alerts \
  arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts
```

ΕΞΟΔΟΣ

```
Error: resource address "aws_sns_topic.alerts" does not exist in the configuration.

Before importing this resource, please create its configuration in the root module. For example:

resource "aws_sns_topic" "alerts" {
  # (resource arguments)
}
```

Η εντολή δεν γράφει configuration. Ζητά το block και σου δείχνει ένα άδειο παράδειγμα. Γράψε το block για το topic σε νέο αρχείο `topic.tf`. Το Terraform διαβάζει όλα τα αρχεία `.tf` του φακέλου (κεφάλαιο 2).

topic.tf

```
resource "aws_sns_topic" "alerts" {
  name = "tfbook-ch22-alerts"

  tags = {
    team = "ops"
  }
}
```

Δοκίμασε πρώτα με το όνομα του topic ως id.

TERMINAL

```
terraform import aws_sns_topic.alerts tfbook-ch22-alerts
```

ΕΞΟΔΟΣ

```
aws_sns_topic.alerts: Importing from ID "tfbook-ch22-alerts"...
╷
│ Error: could not parse import ID "tfbook-ch22-alerts" as ARN: arn: invalid prefix
│
│
```

Ο provider δεν μπόρεσε να διαβάσει το `tfbook-ch22-alerts` ως ARN. Όπως έλεγε ο πίνακας, το id ενός SNS topic είναι το ARN του. Δώσε το ARN.

TERMINAL

```
terraform import aws_sns_topic.alerts \
  arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts
```

ΕΞΟΔΟΣ

```
aws_sns_topic.alerts: Importing from ID "arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts"...
aws_sns_topic.alerts: Import prepared!
  Prepared aws_sns_topic for import
aws_sns_topic.alerts: Refreshing state... [id=arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

Το `Import successful!` ήρθε χωρίς plan και χωρίς `yes`. Το κείμενο από κάτω λέει ότι το resource είναι ήδη στο state. Μόνο τώρα μπορείς να δεις αν το block ταιριάζει με το topic.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_sns_topic.alerts: Refreshing state... [id=arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Εδώ ταίριαζε. Αν δεν ταίριαζε, θα το μάθαινες μετά την αλλαγή του state και όχι πριν. Το block `import` σου δείχνει τις διαφορές πριν αγγίξει το state.

Αν το resource έχει `count` ή `for_each`, η διεύθυνση περιέχει το index ή το key. Στο shell τη βάζεις σε μονά εισαγωγικά, ώστε να μη χαθούν τα διπλά.

TERMINAL

```
terraform import 'aws_s3_bucket.team["reports"]' tfbook-ch22-reports
```

	Block `import`	`terraform import`
Πού ζει	Στη configuration, περνά από Git και review	Στη γραμμή εντολών, δεν αφήνει ίχνος στον κώδικα
Πότε γίνεται	Στο apply, μετά από plan που το δείχνει	Αμέσως, χωρίς plan και χωρίς ερώτηση
Block `resource`	Γραμμένο ή παραγόμενο με `-generate-config-out`	Πρέπει να υπάρχει πριν
Πόσα resources	Πολλά, με πολλά blocks ή με `for_each`	Ένα σε κάθε εκτέλεση
Διαφορές με τη configuration	Στο ίδιο plan με το import	Στο επόμενο plan, αφού γραφτεί το state
Έκδοση	Από την 1.5, το `for_each` από την 1.7	Από πάντα

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η εξέταση ρωτά ακριβώς αυτή τη διαφορά. Το `terraform import` θέλει γραμμένο block `resource`, γράφει το state αμέσως χωρίς plan, φέρνει ένα resource τη φορά και δεν παράγει configuration. Το block `import` είναι δηλωτικό, φαίνεται στο plan, εκτελείται στο apply, φέρνει πολλά resources μαζί και μπορεί να παράγει configuration. Η HashiCorp προτείνει το block.

Πολλά resources με for_each

Όταν τα αντικείμενα είναι πολλά και όμοια, ένα block `import` ανά αντικείμενο γίνεται κουραστικό. Από την 1.7 το block `import` δέχεται `for_each`, όπως ένα resource (κεφάλαιο 11). Φτιάξε τρία buckets με ένα loop του bash.



TERMINAL

```
for name in reports uploads backups; do
  aws s3api create-bucket --bucket "tfbook-ch22-$name" \
    --create-bucket-configuration LocationConstraint=eu-central-1
done
```

TERMINAL

```
cd ..
mkdir team
cd team
cp ../legacy/.terraform.lock.hcl .
```

Δοκίμασε πρώτα να αφήσεις το Terraform να γράψει και το block `resource`.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

locals {
  buckets = toset(["reports", "uploads", "backups"])
}

import {
  for_each = local.buckets
  to       = aws_s3_bucket.team[each.key]
  id       = "tfbook-ch22-${each.key}"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan -generate-config-out=generated.tf
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid import 'to' expression
│
│   on main.tf line 20, in import:
│   20:   to       = aws_s3_bucket.team[each.key]
│
│ The target resource does not use for_each.
╵
```

Το `-generate-config-out` δεν γράφει resources με `for_each`. Το σφάλμα λέει ότι το resource στόχος δεν χρησιμοποιεί `for_each`, γιατί δεν υπάρχει καν. Το block `resource` το γράφεις εσύ, με το ίδιο set, στο αρχείο `buckets.tf`.

buckets.tf

```
resource "aws_s3_bucket" "team" {
  for_each = local.buckets

  bucket = "tfbook-ch22-${each.key}"
}
```

Για κάθε στοιχείο του `local.buckets` το block `import` παράγει ένα import. Το `each.key` δίνει το στοιχείο, το `to` το χρησιμοποιεί ως key της διεύθυνσης και το `id` φτιάχνει από αυτό το όνομα του bucket. Το resource χρησιμοποιεί το ίδιο set, οπότε οι διευθύνσεις ταιριάζουν ένα προς ένα.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.team["backups"] will be imported
…
  # aws_s3_bucket.team["reports"] will be imported
…
  # aws_s3_bucket.team["uploads"] will be imported
…
Plan: 3 to import, 0 to add, 0 to change, 0 to destroy.
…
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 3 imported, 0 added, 0 changed, 0 destroyed.
```

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.team["backups"]
aws_s3_bucket.team["reports"]
aws_s3_bucket.team["uploads"]
```

Τρία imports από ένα block. Πριν από την 1.7 θα έγραφες τρία blocks `import`, ένα για κάθε key. Το ίδιο σχήμα δουλεύει και με map, όπου το `each.value` μπορεί να δώσει το id.

Import σε resource μέσα σε module

Το block `import` γράφεται στο root module, αλλά ο στόχος του μπορεί να βρίσκεται μέσα σε child module. Στο `to` δίνεις την πλήρη διεύθυνση, με το `module.` και το όνομα του module block μπροστά, όπως στο κεφάλαιο 15. Φτιάξε ένα ακόμη bucket με το CLI και έναν φάκελο με ένα μικρό module.

TERMINAL

```
aws s3api create-bucket --bucket tfbook-ch22-media \
  --create-bucket-configuration LocationConstraint=eu-central-1
```

TERMINAL

```
cd ..
mkdir app
cd app
cp ../legacy/.terraform.lock.hcl .
```

modules/store/main.tf

```
variable "name" {
  type = string
}

resource "aws_s3_bucket" "this" {
  bucket = var.name
}
```

Το root module καλεί το module και έχει το block `import`.

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

module "media" {
  source = "./modules/store"
  name   = "tfbook-ch22-media"
}

import {
  to = module.media.aws_s3_bucket.this
  id = "tfbook-ch22-media"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # module.media.aws_s3_bucket.this will be imported
…
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
…
```

Το plan βρήκε το bucket και θα το γράψει στη διεύθυνση `module.media.aws_s3_bucket.this`. Η εντολή δέχεται την ίδια διεύθυνση, για παράδειγμα `terraform import module.media.aws_s3_bucket.this tfbook-ch22-media`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Στο Terraform 1.12 της εξέτασης τα blocks `import` επιτρέπονται μόνο στο root module. Ένα block `import` στα αρχεία ενός child module σταματά το plan με `Import blocks are only allowed in the root module`. Για resource μέσα σε module γράφεις το block στο root module, με `to` όπως το `module.media.aws_s3_bucket.this`. Από την 1.16 το Terraform δέχεται blocks `import` και μέσα σε child modules. Αυτό είναι νεότερο και εκτός ύλης.

Μετά το import

Το block import μένει ή φεύγει

Μετά το apply το block `import` έχει κάνει τη δουλειά του. Μπορείς να το σβήσεις ή να το αφήσεις. Αν μείνει, το Terraform βλέπει ότι η διεύθυνση είναι ήδη στο state και δεν κάνει τίποτα. Το είδες στον φάκελο `legacy`, όπου το plan μετά το apply έλεγε `No changes` με το `import.tf` ακόμη εκεί. Πολλές ομάδες τα κρατούν για λίγο ως καταγραφή στο Git και τα σβήνουν αργότερα. Σβήσε το τώρα.

TERMINAL

```
cd ../legacy
rm import.tf
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.legacy: Refreshing state... [id=tfbook-ch22-legacy]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το resource έμεινε στο state και το plan δεν έχει αλλαγές. Το block `import` χρειάζεται μόνο μέχρι να γίνει το import.

Το import δεν αλλάζει το αντικείμενο

Το import μόνο διαβάζει το αντικείμενο και γράφει το state. Δεν αλλάζει tags, δεν αλλάζει ρυθμίσεις και δεν ξαναφτιάχνει τίποτα. Η βοήθεια της εντολής `terraform import` το λέει ρητά. `This command will not modify your infrastructure`. Η αλλαγή που είδες στο πρώτο plan του `legacy`, το σβήσιμο του `owner`, δεν ήταν μέρος του import. Ήταν ένα κανονικό update που προέκυψε επειδή η configuration διέφερε από το bucket.

Μετά το import όμως το Terraform διαχειρίζεται το αντικείμενο πλήρως. Μια αλλαγή στη configuration θα αλλάξει το bucket. Ένα `terraform destroy` θα σβήσει ένα bucket που το Terraform δεν έφτιαξε ποτέ.

> **ΠΑΓΙΔΑ**
> Πριν κάνεις import, βεβαιώσου ότι το αντικείμενο πρέπει πια να ακολουθεί τον κύκλο ζωής της configuration σου. Αν το σβήσεις από τη configuration ή κάνεις destroy, θα σβηστεί και στο AWS, ακόμη κι αν το έφτιαξε κάποιος άλλος πριν από χρόνια.

Δύο states για ένα resource

Το Terraform σε προστατεύει από διπλό import στο ίδιο state. Ξαναδοκίμασε το import του topic στον φάκελο `alerts`.

TERMINAL

```
cd ../alerts
terraform import aws_sns_topic.alerts \
  arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts
```

ΕΞΟΔΟΣ

```
aws_sns_topic.alerts: Importing from ID "arn:aws:sns:eu-central-1:123456789012:tfbook-ch22-alerts"...
aws_sns_topic.alerts: Import prepared!
  Prepared aws_sns_topic for import

│ Error: Resource already managed by Terraform
│
│ Terraform is already managing a remote object for aws_sns_topic.alerts. To
│ import to this address you must first remove the existing object from the
│ state.
```

Η διεύθυνση έχει ήδη αντικείμενο, οπότε η εντολή σταματά με `Resource already managed by Terraform`. Ένα άλλο state όμως δεν το βλέπει. Κάθε state ξέρει μόνο τον εαυτό του (κεφάλαιο 4). Φτιάξε μια δεύτερη configuration που περιγράφει το bucket του `legacy` με άλλο όνομα.

TERMINAL

```
cd ..
mkdir second
cd second
cp ../legacy/.terraform.lock.hcl .
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_s3_bucket" "logs" {
  bucket = "tfbook-ch22-legacy"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform import aws_s3_bucket.logs tfbook-ch22-legacy
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.logs: Importing from ID "tfbook-ch22-legacy"...
aws_s3_bucket.logs: Import prepared!
…
Import successful!
…
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.logs will be updated in-place
…
      ~ tags                         = {
          - "owner" = "maria" -> null
          - "team"  = "ops" -> null
…
      ~ tags_all                     = {
          - "owner" = "maria" -> null
          - "team"  = "ops" -> null
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

Το import πέτυχε χωρίς καμία προειδοποίηση. Τώρα δύο configurations πιστεύουν ότι το bucket είναι δικό τους. Η δεύτερη δεν έχει tags, οπότε το plan της θέλει να σβήσει και τα `owner` και `team`. Αν γίνει αυτό το apply, το επόμενο plan του `legacy` θα θέλει να τα ξαναβάλει. Οι δύο configurations θα αναιρούν η μία την άλλη επ' αόριστον. Ακόμη χειρότερα, ένα destroy σε οποιονδήποτε από τους δύο φακέλους σβήνει το bucket και για τους δύο. Το ίδιο ισχύει όταν ο άλλος ιδιοκτήτης είναι CloudFormation ή κάποιο άλλο εργαλείο.

Πριν από ένα import έλεγξε ποιος διαχειρίζεται ήδη το αντικείμενο. Βοηθά πολύ ένα tag σε κάθε resource που λέει ποια configuration το φτιάχνει, για παράδειγμα `ManagedBy`, γραμμένο μία φορά στο `default_tags` του κεφαλαίου 17. Εδώ διόρθωσε το λάθος βγάζοντας το bucket από το δεύτερο state με το `terraform state rm` του κεφαλαίου 21. Το bucket μένει στο AWS και στο state του `legacy`.

TERMINAL

```
terraform state rm aws_s3_bucket.logs
```

ΕΞΟΔΟΣ

```
Removed aws_s3_bucket.logs
Successfully removed 1 resource instance(s).
```

ΚΡΑΤΑ ΑΥΤΟ

Ένα πραγματικό αντικείμενο αντιστοιχεί σε μία διεύθυνση σε ένα μόνο state. Το Terraform ελέγχει τη διεύθυνση μέσα στο ίδιο state, αλλά δεν ξέρει τι υπάρχει σε άλλα states. Αυτό το ελέγχεις εσύ πριν από κάθε import.

> **ΣΗΜΕΙΩΣΗ**
> Από την 1.12 το block `import` δέχεται, αντί για `id`, το argument `identity`. Είναι ένα object με τα πεδία που ταυτοποιούν το αντικείμενο. Πρέπει να το υποστηρίζει ο provider για τον συγκεκριμένο τύπο. Ο aws 6.67.0 το υποστηρίζει για το `aws_s3_bucket` με το πεδίο `bucket`, όπως στο απόσπασμα από κάτω. Τα `id` και `identity` δεν μπαίνουν μαζί στο ίδιο block. Στο βιβλίο χρησιμοποιούμε το `id`, που δουλεύει σε κάθε resource.

HCL

```
import {
  to = aws_s3_bucket.legacy
  identity = {
    bucket = "tfbook-ch22-legacy"
  }
}
```

Η ροή σε μία σελίδα

1. Βρες το id του αντικειμένου. Η μορφή του είναι στην ενότητα Import της τεκμηρίωσης του resource.
2. Έλεγξε ότι κανένα άλλο state δεν διαχειρίζεται ήδη το αντικείμενο.
3. Γράψε στο root module ένα block `import` με `to` και `id`, ή με `for_each` για πολλά.
4. Γράψε το block `resource` ή φτιάξε προσχέδιο με `terraform plan -generate-config-out=generated.tf` και καθάρισέ το.
5. Τρέξε plan μέχρι η περίληψη να λέει μόνο `to import`.
6. Κάνε apply και επιβεβαίωσε με ένα plan χωρίς αλλαγές.
7. Σβήσε το block `import` όποτε θέλεις.

Ασκήσεις

Οι ασκήσεις ξεκινούν από αντίγραφο ενός φακέλου `base`, που έχει μόνο τον provider και το lock file. Φτιάξε τον τώρα.

TERMINAL

```
cd ..
mkdir base
cp legacy/.terraform.lock.hcl base/
```

base/providers.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}
```

Σε κάθε άσκηση γράφεις τα blocks `import` και `resource` στο `main.tf` και τρέχεις `terraform init`. Τα αντικείμενα που κάνεις import τα φτιάχνεις πρώτα με το `aws` CLI, ως μέρος της λύσης. Για το apply χρησιμοποίησε `-auto-approve`.

ΑΣΚΗΣΗ 22.1 Ένα topic με block import

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI ένα SNS topic με όνομα `tfbook-ch22-ex1`, χωρίς tags. Φέρε το στο Terraform ως `aws_sns_topic.ex1` με block `import`. Έλεγξε με plan ότι η περίληψη λέει μόνο import και μετά κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Apply complete! Resources: 1 imported, 0 added, 0 changed, 0 destroyed.
```

ΑΣΚΗΣΗ 22.2 Τα tags που δεν ήξερες

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI το bucket `tfbook-ch22-ex2` με τρία tags, `team=data`, `owner=nikos` και `cost=analytics`. Μετά κάνε πως δεν τα θυμάσαι. Γράψε το block `resource` χωρίς tags μαζί με ένα block `import` και τρέξε plan. Βρες τα tags με το `aws` CLI και διόρθωσε τη configuration μέχρι η περίληψη να λέει μόνο import.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 22.3 Configuration από το Terraform

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI το bucket `tfbook-ch22-ex3` με το tag `team=web`. Γράψε μόνο το block `import` προς το `aws_s3_bucket.site` και φτιάξε προσχέδιο με `-generate-config-out`. Μετέφερε το block στο `main.tf`, κράτησε μόνο τα `bucket` και `tags`, σβήσε το παραγόμενο αρχείο και κάνε apply. Το τελευταίο plan πρέπει να μην έχει αλλαγές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
No changes. Your infrastructure matches the configuration.
…
```

ΑΣΚΗΣΗ 22.4 Με την εντολή import

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI το bucket `tfbook-ch22-ex4`, χωρίς tags. Γράψε το block `resource` με διεύθυνση `aws_s3_bucket.ex4` και φέρε το bucket στο state με την εντολή `terraform import`, χωρίς block `import`. Τύπωσε τη λίστα του state.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_s3_bucket.ex4
```

ΑΣΚΗΣΗ 22.5 Τρία parameters με for_each

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με ένα loop του bash τρία SSM parameters τύπου `String`, τα `/tfbook/ch22/ex5/dev`, `/tfbook/ch22/ex5/staging` και `/tfbook/ch22/ex5/prod`, με τιμές `db-dev.internal`, `db-staging.internal` και `db-prod.internal`. Φέρε τα στο Terraform με ένα μόνο block `import` με `for_each` πάνω σε ένα map από local. Το resource λέγεται `aws_ssm_parameter.db_host`. Κάνε apply και τύπωσε τη λίστα του state.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_ssm_parameter.db_host["dev"]
aws_ssm_parameter.db_host["prod"]
aws_ssm_parameter.db_host["staging"]
```

ΑΣΚΗΣΗ 22.6 Ένα id με λάθος

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI το bucket `tfbook-ch22-ex6-archive`. Γράψε block `import` και block `resource`, αλλά στο `id` γράψε κατά λάθος `tfbook-ch22-ex6-archives`. Τρέξε plan και διάβασε το σφάλμα. Μετά διόρθωσε το `id`. Το αναμενόμενο αποτέλεσμα είναι το σφάλμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
╷ Error: Cannot import non-existent remote object
…
│ pre-existing objects can be imported; check that the id is correct and that
│ it is associated with the provider's configured region or endpoint, or use
…
```

ΑΣΚΗΣΗ 22.7 Import και destroy

Ξεκίνα από αντίγραφο του φακέλου `base`. Φτιάξε με το `aws` CLI το bucket `tfbook-ch22-ex7` και φέρε το στο Terraform με block `import`. Κάνε apply και μετά `terraform destroy -auto-approve`. Έλεγξε με `aws s3api head-bucket` αν το bucket υπάρχει ακόμη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws: [ERROR]: An error occurred (404) when calling the HeadBucket operation: Not Found
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 22.1

Μια ομάδα θέλει να φέρει στο Terraform δέκα security groups που φτιάχτηκαν από την κονσόλα. Θέλει να δει τι θα γίνει πριν αλλάξει το state και να περάσει η αλλαγή από code review. Τι πρέπει να χρησιμοποιήσει;

a. Την εντολή `terraform import` για κάθε security group, γιατί δείχνει plan πριν γράψει το state.
b. Blocks `import` στη configuration, που εμφανίζονται στο plan και εκτελούνται στο apply.
c. Το `terraform apply -refresh-only`, που προσθέτει στο state όσα βρίσκει στον λογαριασμό.
d. Το `terraform state push` με ένα state που περιέχει τα security groups.

ΕΡΩΤΗΣΗ 22.2

Τρέχεις `terraform import aws_instance.web i-0abc12345def67890` και παίρνεις το παρακάτω σφάλμα. Τι λείπει;

```
resource address "aws_instance.web" does not exist in the configuration
```

a. Ένα block `import` με `to = aws_instance.web`.
b. Η σημαία `-generate-config-out` στην εντολή `terraform import`.
c. Ένα block `resource "aws_instance" "web"` στη configuration.
d. Ένα `terraform init -upgrade` για νεότερο provider.

ΕΡΩΤΗΣΗ 22.3

Πού βρίσκεις τη μορφή του id που δέχεται ένα συγκεκριμένο resource type για import;

a. Στην τεκμηρίωση του resource στον provider, στην ενότητα Import.
b. Στο `terraform.tfstate` του φακέλου εργασίας.
c. Στην έξοδο της εντολής `terraform providers`.
d. Πουθενά, γιατί είναι πάντα το όνομα του block `resource`.

ΕΡΩΤΗΣΗ 22.4

Ποιες δύο προτάσεις για το `terraform plan -generate-config-out=generated.tf` ισχύουν; Επίλεξε δύο.

a. Γράφει blocks `resource` μόνο για τα blocks `import` που δεν έχουν ήδη block `resource`.
b. Σταματά με σφάλμα αν το `generated.tf` υπάρχει ήδη.
c. Κάνει το import στο state χωρίς apply.
d. Παράγει πάντα configuration που δίνει plan χωρίς σφάλματα.
e. Βρίσκει μόνο του όλα τα resources του λογαριασμού, χωρίς blocks `import`.

ΕΡΩΤΗΣΗ 22.5

Σωστό ή λάθος; Το import αλλάζει το πραγματικό αντικείμενο ώστε να ταιριάζει με τη configuration.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 22.6

Ένας συνάδελφος διαχειρίζεται ένα S3 bucket με τη δική του configuration και το δικό του state. Κάνεις import το ίδιο bucket στη δική σου configuration, που έχει άλλο state. Τι συμβαίνει;

a. Το Terraform αρνείται με `Resource already managed by Terraform`.
b. Το bucket μεταφέρεται από το state του συναδέλφου στο δικό σου.
c. Το Terraform κλειδώνει το bucket ώστε να το αλλάζει μόνο το ένα state.
d. Το import πετυχαίνει και οι δύο configurations διαχειρίζονται πια το ίδιο bucket, οπότε μπορούν να αναιρούν η μία τις αλλαγές της άλλης ή να το σβήσει η μία.



ΚΕΦΑΛΑΙΟ 23

Refactoring με moved και removed

Ο κώδικας του Terraform αλλάζει μορφή όπως κάθε κώδικας. Μετονομάζεις resources, τα μαζεύεις σε modules, περνάς από `count` σε `for_each`. Σε αυτό το κεφάλαιο βλέπεις γιατί μια τέτοια αλλαγή μπορεί να σβήσει πραγματική υποδομή και πώς τα blocks `moved` και `removed` την κάνουν ασφαλή, με plan που ελέγχεις πριν από το apply.

Η διεύθυνση είναι η ταυτότητα

Refactoring λέγεται η αλλαγή στη δομή του κώδικα που δεν αλλάζει τη συμπεριφορά του. Σε μια γλώσσα προγραμματισμού μετονομάζεις μια συνάρτηση, διορθώνεις όσα σημεία την καλούν και το πρόγραμμα κάνει ό,τι έκανε και πριν. Στο Terraform υπάρχει μια επιπλέον δυσκολία, το state.

Στο κεφάλαιο 4 είδες ότι το state συνδέει κάθε resource της configuration με ένα πραγματικό αντικείμενο. Το κλειδί αυτής της σύνδεσης είναι η **διεύθυνση** (address) του resource, για παράδειγμα `aws_s3_bucket.b`. Η διεύθυνση έχει τον τύπο και το όνομα του resource. Μέσα σε child module ξεκινά με `module.` και το όνομα του module block, όπως στο κεφάλαιο 15. Με `count` και `for_each` τελειώνει σε `[0]` ή σε `["logs"]`, όπως στο κεφάλαιο 11.

Το Terraform δεν ξέρει την ιστορία του κώδικά σου. Σε κάθε plan έχει μπροστά του δύο λίστες διευθύνσεων, όσες υπάρχουν στο state και όσες υπάρχουν στη configuration. Μια διεύθυνση που υπάρχει μόνο στο state σημαίνει destroy. Μια διεύθυνση που υπάρχει μόνο στη configuration σημαίνει create. Αν αλλάξεις τη διεύθυνση ενός resource, για το Terraform ένα resource έφυγε και ένα άλλο ήρθε.

Το πρόβλημα

Φτιάξε τον φάκελο του παραδείγματος. Αυτή τη φορά το block `terraform` και ο provider μπαίνουν στο δικό τους αρχείο, το `providers.tf`. Έτσι το `main.tf` έχει μόνο όσα αλλάζουν στο κεφάλαιο.

TERMINAL

```
mkdir reports
cd reports
```

providers.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}
```

main.tf

```
resource "aws_s3_bucket" "b" {
  bucket = "tfbook-ch23-reports"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το bucket θα κρατά αναφορές. Ανέβασε ένα αρχείο με το `aws` CLI του κεφαλαίου 17. Το `-` στη θέση του τοπικού αρχείου σημαίνει ότι το περιεχόμενο έρχεται από την είσοδο της εντολής.

TERMINAL

```
echo "quarter,total" | aws s3 cp - s3://tfbook-ch23-reports/q3.csv
aws s3 ls s3://tfbook-ch23-reports
```

ΕΞΟΔΟΣ

```
2026-10-08 05:35:13         14 q3.csv
```

Το όνομα `b` δεν λέει τίποτα. Κάποιος στην ομάδα το αλλάζει σε `reports`, που περιγράφει τι είναι το bucket. Κανένα argument δεν αλλάζει.

main.tf

```
resource "aws_s3_bucket" "reports" {
  bucket = "tfbook-ch23-reports"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.b will be destroyed
  # (because aws_s3_bucket.b is not in configuration)
…
  # aws_s3_bucket.reports will be created
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

Το plan θέλει να σβήσει το `aws_s3_bucket.b` με την αιτιολογία `because aws_s3_bucket.b is not in configuration` και να φτιάξει το `aws_s3_bucket.reports`. Η περίληψη λέει `1 to add` και `1 to destroy`. Το Terraform δεν καταλαβαίνει ότι πρόκειται για το ίδιο bucket με άλλο όνομα στον κώδικα. Βλέπει μια διεύθυνση στο state χωρίς block στη configuration και ένα block χωρίς διεύθυνση στο state.

Αν κάνεις apply, το Terraform θα προσπαθήσει να σβήσει το bucket και να φτιάξει ένα καινούργιο, άδειο, με το ίδιο όνομα. Εδώ σε σώζει μια σύμπτωση. Το AWS δεν σβήνει bucket που έχει αρχεία, οπότε το apply θα αποτύγχανε. Ένας server δεν έχει τέτοια προστασία. Θα αντικαθίστατο κανονικά και ό,τι είχε μέσα θα χανόταν.

> **ΠΑΓΙΔΑ**
> Σε ένα pull request η μετονομασία μοιάζει αθώα. Ο reviewer βλέπει ένα όνομα να αλλάζει και εγκρίνει. Μόνο το plan δείχνει ότι η αλλαγή σημαίνει destroy. Μετά από κάθε refactoring διάβαζε την περίληψη του plan πριν από οτιδήποτε άλλο.

Το block moved

Το block `moved` λέει στο Terraform ότι ένα αντικείμενο που είναι γραμμένο στο state με μια διεύθυνση ανήκει πλέον σε άλλη. Είναι block πρώτου επιπέδου, όπως το `resource`, δεν έχει labels και έχει δύο arguments.

- Το `from` είναι η παλιά διεύθυνση, όπως είναι γραμμένη στο state.
- Το `to` είναι η νέα διεύθυνση, όπως είναι γραμμένη στη configuration.

Οι διευθύνσεις γράφονται χωρίς εισαγωγικά, όπως μια αναφορά. Πρόσθεσε το block στο `main.tf`, κάτω από το resource.

main.tf

```
resource "aws_s3_bucket" "reports" {
  bucket = "tfbook-ch23-reports"
}

moved {
  from = aws_s3_bucket.b
  to   = aws_s3_bucket.reports
}
```

TERMINAL

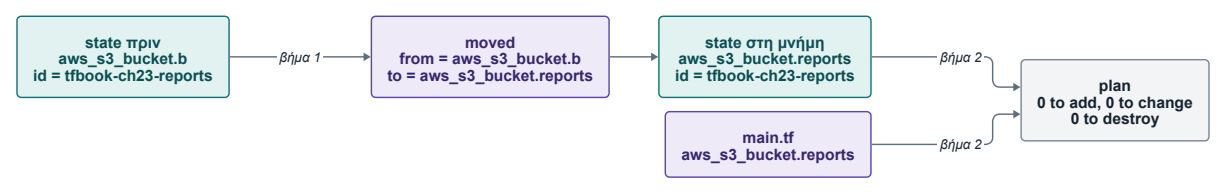
```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.reports: Refreshing state... [id=tfbook-ch23-reports]

Terraform will perform the following actions:

  # aws_s3_bucket.b has moved to aws_s3_bucket.reports
    resource "aws_s3_bucket" "reports" {
        id                          = "tfbook-ch23-reports"
        tags                        = {}
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η πρώτη γραμμή δείχνει ότι το refresh έγινε ήδη με τη νέα διεύθυνση, `aws_s3_bucket.reports`. Το Terraform εφαρμόζει πρώτα τα blocks `moved` στο αντίγραφο του state που κρατά στη μνήμη. Μετά συγκρίνει.

Η γραμμή `aws_s3_bucket.b has moved to aws_s3_bucket.reports` δεν έχει κανένα σύμβολο μπροστά, γιατί δεν είναι ενέργεια στο AWS. Από κάτω το Terraform δείχνει το resource όπως είναι, με `id` το `tfbook-ch23-reports`, για να βεβαιωθείς ότι μετακινείται το σωστό αντικείμενο. Η περίληψη είναι `0 to add, 0 to change, 0 to destroy`.



Σχήμα 23.1 · Το moved εφαρμόζεται πριν από τη σύγκριση. Το Terraform αλλάζει τη διεύθυνση στο state που κρατά στη μνήμη και μετά τη συγκρίνει με τη configuration. Το id μένει ίδιο, άρα και το bucket.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Το apply γράφει το νέο state χωρίς να αλλάξει τίποτα στο AWS. Η περίληψη το επιβεβαιώνει. Δες τι κρατά τώρα το state και αν το αρχείο είναι ακόμη στη θέση του.

TERMINAL

```
terraform state list
aws s3 ls s3://tfbook-ch23-reports
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.reports
2026-10-08 05:35:13         14 q3.csv
```

Το state έχει μόνο τη νέα διεύθυνση. Το `q3.csv` είναι εκεί, γιατί το bucket είναι το ίδιο. Τρέξε ξανά plan με το block `moved` ακόμη στο αρχείο.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

Το block δεν πειράζει. Το `from` δεν υπάρχει πια στο state, οπότε το Terraform το προσπερνά.

ΚΡΑΤΑ ΑΥΤΟ

Το `moved` αλλάζει μόνο τη διεύθυνση ενός αντικειμένου στο state. Δεν καλεί το API, δεν αλλάζει arguments και δεν αγγίζει το πραγματικό resource. Ένα σωστό refactoring δίνει plan με `0 to add` και `0 to destroy`.



Οι κανόνες του moved

Για κάθε block `moved` το plan βρίσκεται σε μία από τις παρακάτω περιπτώσεις.

- Αν το `from` υπάρχει στο state και το `to` όχι, μεταφέρει το αντικείμενο.
- Αν το `from` δεν υπάρχει στο state, δεν κάνει τίποτα. Γι' αυτό ένα block που έχει ήδη εφαρμοστεί δεν ενοχλεί.
- Αν το `from` είναι ακόμη δηλωμένο στη configuration, το plan σταματά με σφάλμα `Moved object still exists`. Έγραψες το `moved` αλλά ξέχασες να μετονομάσεις το resource block.
- Αν υπάρχουν και οι δύο διευθύνσεις στο state, δεν μεταφέρει. Βγάζει την προειδοποίηση `Unresolved resource instance address changes` και το plan σβήνει το παλιό αντικείμενο.
- Αν γράψεις λάθος το `to`, η μεταφορά γίνεται σε διεύθυνση που δεν υπάρχει στη configuration και το plan σβήνει το αντικείμενο. Μία από τις ασκήσεις το δείχνει.

Το `terraform validate` δεν πιάνει αυτά τα λάθη. Φαίνονται στο plan, που είναι ο λόγος να το διαβάζεις.

Μπορείς επίσης να μετονομάσεις ένα resource και να αλλάξεις ένα argument του στο ίδιο commit. Το Terraform κάνει πρώτα τη μεταφορά και μετά τη σύγκριση. Το plan γράφει τότε `will be updated in-place` και από κάτω `(moved from aws_s3_bucket.b)`, με τις διαφορές του argument όπως σε κάθε update.

Μεταφορά σε module

Το bucket των αναφορών θα αποκτήσει κι άλλες ρυθμίσεις και η ομάδα αποφασίζει να το βάλει σε module, όπως έμαθες στο κεφάλαιο 15. Γράψε το module.

modules/bucket/main.tf

```
variable "name" {
  type = string
}

resource "aws_s3_bucket" "this" {
  bucket = var.name
}

output "arn" {
  value = aws_s3_bucket.this.arn
}
```

Στο root module το resource γίνεται κλήση του module. Η νέα διεύθυνση του bucket είναι `module.store.aws_s3_bucket.this`, δηλαδή το όνομα του module block και μετά η διεύθυνση του resource μέσα στο module.

main.tf

```
module "store" {
  source = "./modules/bucket"
  name   = "tfbook-ch23-reports"
}

moved {
  from = aws_s3_bucket.b
  to   = aws_s3_bucket.reports
}

moved {
  from = aws_s3_bucket.reports
  to   = module.store.aws_s3_bucket.this
}
```

Το νέο `moved` μπαίνει στο root module, γιατί μόνο αυτό βλέπει και τις δύο διευθύνσεις. Το παλιό μένει. Θα δεις σε λίγο γιατί. Το module block είναι καινούργιο, οπότε χρειάζεται `terraform init`.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
Initializing modules...
- store in modules/bucket
…
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
module.store.aws_s3_bucket.this: Refreshing state... [id=tfbook-ch23-reports]

Terraform will perform the following actions:

  # aws_s3_bucket.reports has moved to module.store.aws_s3_bucket.this
    resource "aws_s3_bucket" "this" {
        id                          = "tfbook-ch23-reports"
        tags                        = {}
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Η γραμμή `aws_s3_bucket.reports has moved to module.store.aws_s3_bucket.this` λέει ότι το bucket μπαίνει στο module χωρίς να ξαναφτιαχτεί. Το resource εμφανίζεται πλέον με το όνομα που έχει μέσα στο module, `this`. Η περίληψη μένει `0 to add, 0 to change, 0 to destroy`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Μετονομασία module

Το όνομα `store` δεν λέει τι κρατά το module. Η ομάδα το αλλάζει σε `reports`. Ένα `moved` με διευθύνσεις module μεταφέρει όλα τα resources του module μαζί, όσα κι αν είναι.

main.tf

```
module "reports" {
  source = "./modules/bucket"
  name   = "tfbook-ch23-reports"
}

moved {
  from = aws_s3_bucket.b
  to   = aws_s3_bucket.reports
}

moved {
  from = aws_s3_bucket.reports
  to   = module.store.aws_s3_bucket.this
}

moved {
  from = module.store
  to   = module.reports
}
```

Το module block έχει νέο όνομα, οπότε το `init` χρειάζεται ξανά.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- reports in modules/bucket
…
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
module.reports.aws_s3_bucket.this: Refreshing state... [id=tfbook-ch23-reports]

Terraform will perform the following actions:

  # module.store.aws_s3_bucket.this has moved to module.reports.aws_s3_bucket.this
    resource "aws_s3_bucket" "this" {
        id                          = "tfbook-ch23-reports"
        tags                        = {}
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Το plan γράφει `module.store.aws_s3_bucket.this has moved to module.reports.aws_s3_bucket.this`. Ένα module με δέκα resources θα έδειχνε δέκα τέτοιες γραμμές από ένα μόνο block.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
module.reports.aws_s3_bucket.this
```

Αλυσίδα από moved

Τα τρία blocks σχηματίζουν αλυσίδα. Το `b` έγινε `reports`, το `reports` έγινε `module.store.aws_s3_bucket.this` και το `module.store` έγινε `module.reports`. Το Terraform ακολουθεί την αλυσίδα. Ένα state που έχει ακόμη την πιο παλιά διεύθυνση πηγαίνει κατευθείαν στην τελευταία.

Για να το δεις, γύρισε προσωρινά το state στη διεύθυνση `aws_s3_bucket.b` με την εντολή `terraform state mv` του κεφαλαίου 21. Έτσι μοιάζει με ένα δεύτερο state του ίδιου κώδικα, για παράδειγμα του production, που δεν έχει περάσει από κανένα από τα τρία apply.

TERMINAL

```
terraform state mv module.reports.aws_s3_bucket.this aws_s3_bucket.b
```

ΕΞΟΔΟΣ

```
Move "module.reports.aws_s3_bucket.this" to "aws_s3_bucket.b"
Successfully moved 1 object(s).
```

Πρόσεξε τη διαφορά από το `moved`. Η εντολή άλλαξε το state αμέσως, χωρίς plan και χωρίς ερώτηση. Τώρα τρέξε plan.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.b has moved to module.reports.aws_s3_bucket.this
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

Μία γραμμή, `aws_s3_bucket.b has moved to module.reports.aws_s3_bucket.this`. Το Terraform ένωσε τα τρία βήματα σε ένα. Αν είχες σβήσει τα δύο πρώτα blocks μετά το δικό σου apply, το ίδιο state θα έβγαζε plan με destroy και create. Κάνε apply για να επιστρέψει το state στη θέση του.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Από count σε for_each

Στο κεφάλαιο 11 είδες ότι τα instances του `count` αναγνωρίζονται από τη θέση τους. Αν σβήσεις ένα στοιχείο από τη μέση της λίστας, όσα ακολουθούν αλλάζουν θέση και αντικαθίστανται. Ένα resource που ήδη υπάρχει με `count` γυρίζει σε `for_each` μόνο με τη βοήθεια του `moved`.

Φτιάξε νέο φάκελο. Αντέγραψε το `providers.tf` και το lock file. Όπως είδες στο κεφάλαιο 17, με το lock file το init παίρνει τον provider από το plugin cache αντί να τον κατεβάσει ξανά.

TERMINAL

```
cd ..
mkdir buckets
cp reports/providers.tf reports/.terraform.lock.hcl buckets/
cd buckets
```

main.tf

```
variable "names" {
  type    = list(string)
  default = ["logs", "backups"]
}

resource "aws_s3_bucket" "b" {
  count  = length(var.names)
  bucket = "tfbook-ch23-${var.names[count.index]}"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

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

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.b[0]
aws_s3_bucket.b[1]
```

Γύρισε το resource σε `for_each` με το `toset` του κεφαλαίου 11.

main.tf

```
variable "names" {
  type    = list(string)
  default = ["logs", "backups"]
}

resource "aws_s3_bucket" "b" {
  for_each = toset(var.names)
  bucket   = "tfbook-ch23-${each.key}"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.b[0] will be destroyed
  # (because resource does not use count)
…
  # aws_s3_bucket.b[1] will be destroyed
  # (because resource does not use count)
…
  # aws_s3_bucket.b["backups"] will be created
…
  # aws_s3_bucket.b["logs"] will be created
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Τα δύο buckets σβήνονται `because resource does not use count` και ξαναφτιάχνονται με κλειδιά. Η περίληψη λέει `2 to add` και `2 to destroy`. Τα ονόματα των buckets δεν άλλαξαν καθόλου. Άλλαξαν μόνο οι διευθύνσεις.

Χρειάζεται ένα `moved` για κάθε instance. Στο `from` γράφεις τον αριθμό της θέσης και στο `to` το κλειδί σε εισαγωγικά, ακριβώς όπως στις διευθύνσεις του κεφαλαίου 11.

main.tf

```
variable "names" {
  type    = list(string)
  default = ["logs", "backups"]
}

resource "aws_s3_bucket" "b" {
  for_each = toset(var.names)
  bucket   = "tfbook-ch23-${each.key}"
}

moved {
  from = aws_s3_bucket.b[0]
  to   = aws_s3_bucket.b["logs"]
}

moved {
  from = aws_s3_bucket.b[1]
  to   = aws_s3_bucket.b["backups"]
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.b["backups"]: Refreshing state... [id=tfbook-ch23-backups]
aws_s3_bucket.b["logs"]: Refreshing state... [id=tfbook-ch23-logs]

Terraform will perform the following actions:

  # aws_s3_bucket.b[1] has moved to aws_s3_bucket.b["backups"]
    resource "aws_s3_bucket" "b" {
        id                          = "tfbook-ch23-backups"
        tags                        = {}
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

  # aws_s3_bucket.b[0] has moved to aws_s3_bucket.b["logs"]
    resource "aws_s3_bucket" "b" {
        id                          = "tfbook-ch23-logs"
        tags                        = {}
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Δύο μεταφορές, μία για κάθε bucket. Η περίληψη είναι `0 to add, 0 to change, 0 to destroy`. Η αντιστοίχιση θέσης σε κλειδί είναι δική σου ευθύνη. Αν γράψεις `b[0]` προς `b["backups"]`, το Terraform θα

συνδέσει το bucket `tfbook-ch23-logs` με το κλειδί `backups` και το plan θα προτείνει αντικατάσταση, γιατί το `bucket` δεν θα ταιριάζει.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.b["backups"]
aws_s3_bucket.b["logs"]
```

ΣΗΜΕΙΩΣΗ

Για μία περίπτωση το Terraform κάνει τη μεταφορά μόνο του. Αν προσθέσεις `count` σε resource που δεν είχε, το `aws_s3_bucket.x` γίνεται `aws_s3_bucket.x[0]` χωρίς block `moved`. Αν αφαιρέσεις το `count`, το `[0]` γίνεται ξανά διεύθυνση χωρίς index. Το plan το δείχνει πάλι με `has moved to`. Για το `for_each` αυτό δεν γίνεται, γιατί το Terraform δεν μπορεί να μαντέψει το κλειδί.

Πότε σβήνεις τα moved blocks

Ένα `moved` χρειάζεται όσο υπάρχει έστω και ένα state με την παλιά διεύθυνση. Η απάντηση εξαρτάται από το πόσα states χρησιμοποιούν τον κώδικα.

- **Root module με ένα state.** Μετά το apply κανένα state δεν έχει την παλιά διεύθυνση. Μπορείς να σβήσεις τα blocks. Πολλές ομάδες τα κρατούν για λίγο ως σημείωση της αλλαγής.
- **Ίδιος κώδικας σε πολλά states.** Τα workspaces του κεφαλαίου 20 ή χωριστοί φάκελοι για development και production που καλούν το ίδιο module έχουν το καθένα δικό του state. Κάθε state παίρνει τη μεταφορά στο δικό του apply. Κράτα τα blocks μέχρι να περάσουν όλα.
- **Module που το χρησιμοποιούν άλλοι.** Ο συγγραφέας ενός module του Registry, όπως αυτά του κεφαλαίου 16, δεν ξέρει πότε θα αναβαθμίσει ο κάθε χρήστης. Κάποιος μπορεί να πάει από την έκδοση 1.0 στην 3.0 και να χρειαστεί όλες τις ενδιάμεσες μεταφορές. Εδώ τα blocks μένουν. Αν τα σβήσεις, η νέα έκδοση του module γίνεται breaking change.

Μέσα σε ένα module οι διευθύνσεις των `moved` είναι σχετικές με το module. Γράφεις `aws_s3_bucket.this` και όχι `module.reports.aws_s3_bucket.this`, γιατί το module δεν ξέρει με ποιο όνομα θα το καλέσουν.

Στο `buckets` υπάρχει ένα state και έχει ήδη περάσει το apply. Σβήσε τα δύο blocks.

main.tf

```
variable "names" {
  type    = list(string)
  default = ["logs", "backups"]
}

resource "aws_s3_bucket" "b" {
  for_each = toset(var.names)
  bucket   = "tfbook-ch23-${each.key}"
}
```

TERMINAL

```
terraform plan
```

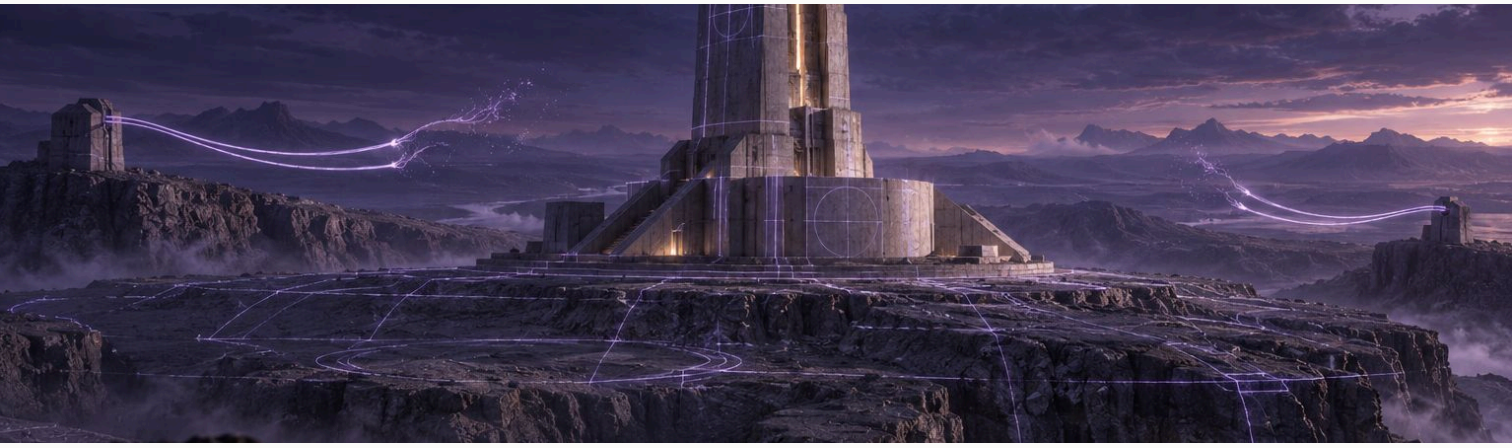
ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

Το block removed

Μερικές φορές θέλεις το Terraform να σταματήσει να διαχειρίζεται ένα resource χωρίς να το σβήσει. Το bucket περνά σε άλλη ομάδα με δική της configuration. Ένα μεγάλο project χωρίζεται στα δύο. Ένα resource θα το διαχειρίζεται στο εξής άλλο εργαλείο. Αν απλώς σβήσεις το resource block, το plan θα σβήσει και το πραγματικό αντικείμενο.

Πριν από το Terraform 1.7 ο μόνος τρόπος ήταν το `terraform state rm` του κεφαλαίου 21. Από την 1.7 υπάρχει το block `removed`. Φτιάξε τον τρίτο φάκελο του κεφαλαίου με δύο buckets.



TERMINAL

```
cd ..
mkdir handover
cp reports/providers.tf reports/.terraform.lock.hcl handover/
cd handover
```

main.tf

```
resource "aws_s3_bucket" "shared" {
  bucket = "tfbook-ch23-shared"
}

resource "aws_s3_bucket" "scratch" {
  bucket = "tfbook-ch23-scratch"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

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

Το `shared` περνά σε άλλη ομάδα. Δες πρώτα τι γίνεται αν σβήσεις απλώς το block του.

main.tf

```
resource "aws_s3_bucket" "scratch" {
  bucket = "tfbook-ch23-scratch"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.shared will be destroyed
  # (because aws_s3_bucket.shared is not in configuration)
…
Plan: 0 to add, 0 to change, 1 to destroy.
…
```

Η αιτιολογία είναι ίδια με της μετονομασίας, `is not in configuration`. Η περίληψη λέει `1 to destroy`. Αυτό δεν είναι παράδοση. Γράψε στη θέση του block ένα `removed`.

main.tf

```
resource "aws_s3_bucket" "scratch" {
  bucket = "tfbook-ch23-scratch"
}

removed {
  from = aws_s3_bucket.shared

  lifecycle {
    destroy = false
  }
}
```

Το `from` είναι η διεύθυνση που φεύγει από το state. Το block `lifecycle` μέσα στο `removed` δέχεται ένα μόνο argument, το `destroy`. Με `false` το Terraform ξεχνά το αντικείμενο αλλά δεν το σβήνει.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.scratch: Refreshing state... [id=tfbook-ch23-scratch]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

 # aws_s3_bucket.shared will no longer be managed by Terraform, but will not be destroyed
 # (destroy = false is set in the configuration)
 . resource "aws_s3_bucket" "shared" {
        id                          = "tfbook-ch23-shared"
        # (15 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.

╷
│ Warning: Some objects will no longer be managed by Terraform
│
│ If you apply this plan, Terraform will discard its tracking information for
│ the following objects, but it will not delete them:
│  - aws_s3_bucket.shared
│
│ After applying this plan, Terraform will no longer manage these objects.
│ You will need to import them into Terraform to manage them again.
╵

──────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Μπροστά από το `resource "aws_s3_bucket" "shared"` υπάρχει μια τελεία. Είναι το σύμβολο του plan για αντικείμενο που φεύγει από το state χωρίς destroy. Το κείμενο λέει ακριβώς τι θα γίνει, `will no longer`

`be managed by Terraform, but will not be destroyed`. Από κάτω γράφει τον λόγο, `destroy = false is set in the configuration`.

Πρόσεξε την περίληψη. Λέει `0 to add, 0 to change, 0 to destroy`, γιατί δεν μετρά τα αντικείμενα που ξεχνιούνται. Αυτό που σε ειδοποιεί είναι η προειδοποίηση `Some objects will no longer be managed by Terraform`. Δίνει τη λίστα των διευθύνσεων και σου θυμίζει ότι για να τα διαχειριστείς ξανά θα χρειαστείς import, όπως στο κεφάλαιο 22.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

TERMINAL

```
terraform state list
aws s3 ls
```

ΕΞΟΔΟΣ

```
aws_s3_bucket.scratch
2026-10-08 05:35:10 tfbook-ch23-reports
2026-10-08 05:35:53 tfbook-ch23-backups
2026-10-08 05:35:53 tfbook-ch23-logs
2026-10-08 05:36:08 tfbook-ch23-shared
2026-10-08 05:36:08 tfbook-ch23-scratch
```

Το state έχει μόνο το `scratch`. Το `aws s3 ls` δείχνει όλα τα buckets του λογαριασμού και ανάμεσά τους το `tfbook-ch23-shared`, που υπάρχει ακόμη. Η λίστα έχει και όλα τα buckets από τις προηγούμενες ενότητες. Κανένα refactoring του κεφαλαίου δεν έσβησε bucket.

Το `removed` έχει τρεις κανόνες που αξίζει να θυμάσαι.

- Το resource block πρέπει να έχει φύγει από τη configuration. Αν μείνει, το plan σταματά με σφάλμα `Removed resource still exists`.
- Το `from` δέχεται ολόκληρο resource ή ολόκληρο module, για παράδειγμα `module.reports`. Δεν δέχεται ένα μόνο instance, όπως `aws_s3_bucket.b["logs"]`.
- Χωρίς block `lifecycle` το Terraform σβήνει το αντικείμενο, σαν να έγραψες `destroy = true`. Γράφε πάντα το `destroy` ρητά.

removed με destroy = true

Το `scratch` δεν χρειάζεται πια και θέλεις να σβηστεί. Μπορείς να σβήσεις το block του, όπως είδες. Μπορείς επίσης να γράψεις την πρόθεσή σου ρητά.

main.tf

```
removed {
  from = aws_s3_bucket.shared

  lifecycle {
    destroy = false
  }
}

removed {
  from = aws_s3_bucket.scratch

  lifecycle {
    destroy = true
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_s3_bucket.scratch will be destroyed
  # (because aws_s3_bucket.scratch is not in configuration)
…
Plan: 0 to add, 0 to change, 1 to destroy.
…
```

Το plan είναι ίδιο με αυτό που θα έβγαζε το σβήσιμο του block, με την ίδια αιτιολογία. Η διαφορά είναι στο pull request. Ο reviewer διαβάζει ότι το destroy είναι σκόπιμο και όχι αποτέλεσμα ενός block που χάθηκε κατά λάθος. Το `removed` για το `shared` δεν βγάζει τίποτα πια, γιατί το `shared` δεν υπάρχει στο state.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.
```

TERMINAL

```
aws s3 ls
```

ΕΞΟΔΟΣ

```
2026-10-08 05:35:10 tfbook-ch23-reports
2026-10-08 05:35:53 tfbook-ch23-backups
2026-10-08 05:35:53 tfbook-ch23-logs
2026-10-08 05:36:08 tfbook-ch23-shared
```

Το `tfbook-ch23-scratch` έφυγε. Το `tfbook-ch23-shared` και τα υπόλοιπα έμειναν.

ΠΑΓΙΔΑ

Μην μπερδέψεις το `removed` με `lifecycle { destroy = false }` με το `lifecycle { destroy = false }` μέσα σε κανονικό resource block. Το δεύτερο προστέθηκε στο Terraform 1.16, το είδες στο κεφάλαιο 14 και είναι εκτός ύλης εξέτασης. Μένει σε resource block που υπάρχει ακόμη και ορίζει τι θα γίνει σε ένα μελλοντικό destroy ή σε αντικατάσταση. Το `removed` λέει ότι το block έφυγε και ξεχνά το αντικείμενο στο επόμενο apply. Το `lifecycle` του `removed` δέχεται μόνο το `destroy`. Ένα `prevent_destroy` εκεί δίνει σφάλμα `Unsupported argument`.

moved και removed απέναντι στις εντολές state

Το `terraform state mv` κάνει ό,τι το `moved` και το `terraform state rm` ό,τι το `removed` με `destroy = false`. Το αποτέλεσμα στο state είναι ίδιο. Ο δρόμος είναι διαφορετικός.

	`state mv`, `state rm`	`moved`, `removed`
Πότε αλλάζει το state	Αμέσως, μόλις τρέξεις την εντολή	Στο apply
Plan πριν από την αλλαγή	Όχι	Ναι
Review σε pull request	Όχι, η εντολή δεν μένει στον κώδικα	Ναι, το block είναι κώδικας
HCP Terraform χωρίς πρόσβαση στο state	Όχι	Ναι, εφαρμόζεται μέσα στο run
Πολλά states με τον ίδιο κώδικα	Μία εντολή για κάθε state	Ένα block για όλα
Ένα μόνο instance εκτός διαχείρισης	Ναι, με `state rm`	Όχι

Οι εντολές state τρέχουν από τον υπολογιστή σου και γράφουν απευθείας στο state. Χρειάζονται δικαίωμα εγγραφής σε αυτό και κανείς άλλος δεν βλέπει την αλλαγή πριν γίνει. Στο HCP Terraform του κεφαλαίου 26 τα runs ξεκινούν από το VCS και η ομάδα συχνά δεν έχει πρόσβαση στο state. Ένα block στον κώδικα περνά από το ίδιο review και το ίδιο plan με κάθε άλλη αλλαγή.

Οι εντολές state έχουν ακόμη θέση. Βγάζουν από τη διαχείριση ένα μόνο instance ενός `for_each`, κάτι που το `removed` δεν κάνει. Χρησιμεύουν και σε επισκευές έκτακτης ανάγκης. Το `moved` επίσης δεν μεταφέρει ένα resource από ένα state σε άλλο. Για να περάσει ένα resource σε άλλη configuration γράφεις `removed` στη μία και `import` στην άλλη.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Για refactoring η HashiCorp προτείνει τα blocks `moved` και `removed` αντί για τα `terraform state mv` και `terraform state rm`, γιατί η αλλαγή φαίνεται στο plan και περνά από review. Το `moved` υπάρχει από το Terraform 1.1 και το `removed` από το 1.7, άρα και τα δύο είναι μέσα στην ύλη. Θυμήσου ότι το `removed` με `destroy = false` αφήνει το αντικείμενο στο cloud και ότι το Terraform δεν το διαχειρίζεται πια μέχρι να γίνει νέο import.

Τα δύο blocks σε μία σελίδα

Θέλεις	Γράφεις
Μετονομασία resource	`moved` με `from = aws_s3_bucket.b` και `to = aws_s3_bucket.reports`
Resource μέσα σε module	`moved` με `to = module.store.aws_s3_bucket.this`
Μετονομασία module	`moved` με `from = module.store` και `to = module.reports`
Από `count` σε `for_each`	Ένα `moved` για κάθε instance, από `b[0]` σε `b["logs"]`
Έξοδος από τη διαχείριση	`removed` με `lifecycle { destroy = false }`
Destroy με ρητή πρόθεση	`removed` με `lifecycle { destroy = true }`

Ασκήσεις

Οι ασκήσεις ξεκινούν από αντίγραφο του φακέλου `base`, που έχει μόνο το `providers.tf` και το lock file. Φτιάξε τον τώρα. Κάθε άσκηση γράφει το δικό της `main.tf` και τρέχει `terraform init`. Τα ονόματα των buckets έχουν τον αριθμό της άσκησης, γιατί όλα τα buckets του κεφαλαίου ζουν στον ίδιο λογαριασμό.

TERMINAL

```
cd ~/tfbook/ch23
mkdir base
cp reports/providers.tf reports/.terraform.lock.hcl base/
```

ΑΣΚΗΣΗ 23.1 Μετονομασία με moved

Ξεκίνα από αντίγραφο του `base`. Γράψε στο `main.tf` ένα `aws_s3_bucket` με όνομα `pics` και bucket `tfbook-ch23-ex1`. Κάνε apply. Μετά μετονόμασε το resource σε `images` χωρίς να σβηστεί το bucket και τρέξε plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.pics has moved to aws_s3_bucket.images
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 23.2 Μετονομασία και αλλαγή μαζί

Ξεκίνα από αντίγραφο του `base`. Φτιάξε το bucket `tfbook-ch23-ex2` με όνομα resource `data` και κάνε apply. Μετά, στην ίδια αλλαγή, μετονόμασε το resource σε `raw` και πρόσθεσε το tag `Stage` με τιμή `raw`. Τι λέει το plan;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.raw will be updated in-place
  # (moved from aws_s3_bucket.data)
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 23.3 Ένα γράμμα λάθος

Ξεκίνα από αντίγραφο του `base`. Φτιάξε το bucket `tfbook-ch23-ex3` με όνομα resource `old` και κάνε apply. Ένας συνάδελφος το μετονομάζει σε `current` και γράφει το παρακάτω block.

HCL

```
moved {
  from = aws_s3_bucket.old
  to   = aws_s3_bucket.curent
}
```

Γράψε την αλλαγή του ακριβώς όπως είναι, τρέξε plan χωρίς apply και βρες στην έξοδο τι θα γινόταν. Μετά διόρθωσε το λάθος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.curent will be destroyed
  # (because aws_s3_bucket.old was moved to aws_s3_bucket.curent, which is not in configuration)
  # (moved from aws_s3_bucket.old)
…
  # aws_s3_bucket.current will be created
…
Plan: 1 to add, 0 to change, 1 to destroy.
…
```

ΑΣΚΗΣΗ 23.4 Δύο resources σε ένα for_each

Ξεκίνα από αντίγραφο του `base`. Φτιάξε δύο resources, το `red` με bucket `tfbook-ch23-ex4-red` και το `blue` με bucket `tfbook-ch23-ex4-blue`. Κάνε apply. Μετά αντικατάστησέ τα με ένα resource `color` που φτιάχνει τα ίδια δύο buckets με `for_each`. Το plan πρέπει να μην αλλάζει τίποτα στο AWS.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # aws_s3_bucket.blue has moved to aws_s3_bucket.color["blue"]
…
  # aws_s3_bucket.red has moved to aws_s3_bucket.color["red"]
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 23.5 moved μέσα σε module

Ξεκίνα από αντίγραφο του `base`. Γράψε το module `modules/box` με μία μεταβλητή `name` και ένα `aws_s3_bucket` με όνομα `main`. Κάλεσέ το από το root module ως `box` με bucket `tfbook-ch23-ex5` και κάνε apply. Μετά, ως συγγραφέας του module, μετονόμασε το resource σε `this`. Βάλε το `moved` μέσα στο module, ώστε οι χρήστες του να μη χρειαστεί να κάνουν τίποτα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # module.box.aws_s3_bucket.main has moved to module.box.aws_s3_bucket.this
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 23.6 Ένα instance εκτός διαχείρισης

Ξεκίνα από αντίγραφο του `base`. Φτιάξε με `for_each` πάνω στο `toset(["a", "b"])` δύο buckets, `tfbook-ch23-ex6-a` και `tfbook-ch23-ex6-b`, σε ένα resource με όνομα `team`. Κάνε apply. Θέλεις να βγει από τη διαχείριση μόνο το bucket του κλειδιού `b`, χωρίς να σβηστεί. Δοκίμασε πρώτα ένα `removed` για το `aws_s3_bucket.team["b"]` και τρέξε `terraform validate`. Μετά βρες τον τρόπο που δουλεύει. Στο τέλος δείξε με `terraform state list` και `aws s3 ls` ότι το state κρατά μόνο το κλειδί `a` και ότι υπάρχουν ακόμη και τα δύο buckets.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
aws_s3_bucket.team["a"]
…
2026-10-08 05:37:00 tfbook-ch23-ex6-a
2026-10-08 05:37:00 tfbook-ch23-ex6-b
```

ΑΣΚΗΣΗ 23.7 Παράδοση σε άλλη configuration

Ξεκίνα από αντίγραφο του `base`. Φτιάξε το bucket `tfbook-ch23-ex7` με όνομα resource `handoff` και κάνε apply. Το bucket περνά σε άλλη ομάδα, που το θέλει στη δική της configuration στον υποφάκελο `other` με όνομα resource `received`. Βγάλε το bucket από αυτή τη configuration χωρίς να σβηστεί και φέρε το στη νέα με block `import` του κεφαλαίου 22. Το αναμενόμενο αποτέλεσμα είναι η περίληψη του plan στο `other`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
…
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 23.1

Μετονομάζεις το resource `aws_instance.web` σε `aws_instance.app`. Θέλεις να μη γίνει αντικατάσταση και η αλλαγή να φαίνεται στο plan και στο code review. Τι κάνεις;

a. Τρέχεις `terraform state mv aws_instance.web aws_instance.app` πριν από το commit.
b. Προσθέτεις block `moved` με `from = aws_instance.web` και `to = aws_instance.app`.
c. Τρέχεις `terraform apply -replace=aws_instance.app`.
d. Τρέχεις `terraform import` για το `aws_instance.app` με το ID του server.

ΕΡΩΤΗΣΗ 23.2

Ένα block `removed` έχει `from = aws_s3_bucket.logs` και `lifecycle { destroy = false }`. Ποιες δύο προτάσεις είναι σωστές; Επίλεξε δύο.

a. Μετά το apply το bucket υπάρχει ακόμη στο AWS.
b. Το `aws_s3_bucket.logs` βγαίνει από το state μόλις τρέξεις `terraform plan`.
c. Το resource block `aws_s3_bucket.logs` πρέπει να έχει αφαιρεθεί από τη configuration.
d. Το `from` μπορεί να είναι και ένα instance, όπως `aws_s3_bucket.logs["eu"]`.
e. Η περίληψη του plan γράφει `1 to destroy`.

ΕΡΩΤΗΣΗ 23.3

Μετά από ένα refactoring το plan δείχνει τα παρακάτω.

```
ΕΞΟΔΟΣ

# aws_s3_bucket.nwe will be destroyed
# (because aws_s3_bucket.old was moved to aws_s3_bucket.nwe, which is not in configuration)
Plan: 1 to add, 0 to change, 1 to destroy.
```

Ποια είναι η πιο πιθανή αιτία;

a. Το resource block έχει ακόμη το παλιό όνομα `old`.
b. Πρέπει πρώτα να τρέξεις `terraform state mv`.
c. Το `to` του block `moved` έχει λάθος στο όνομα.
d. Ο AWS provider δεν υποστηρίζει μεταφορές για S3 buckets.

ΕΡΩΤΗΣΗ 23.4

Σωστό ή λάθος; Ο συγγραφέας ενός module που είναι δημοσιευμένο στο Registry πρέπει να σβήνει τα blocks `moved` στην αμέσως επόμενη έκδοση, αφού η μεταφορά έχει ήδη γίνει.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 23.5

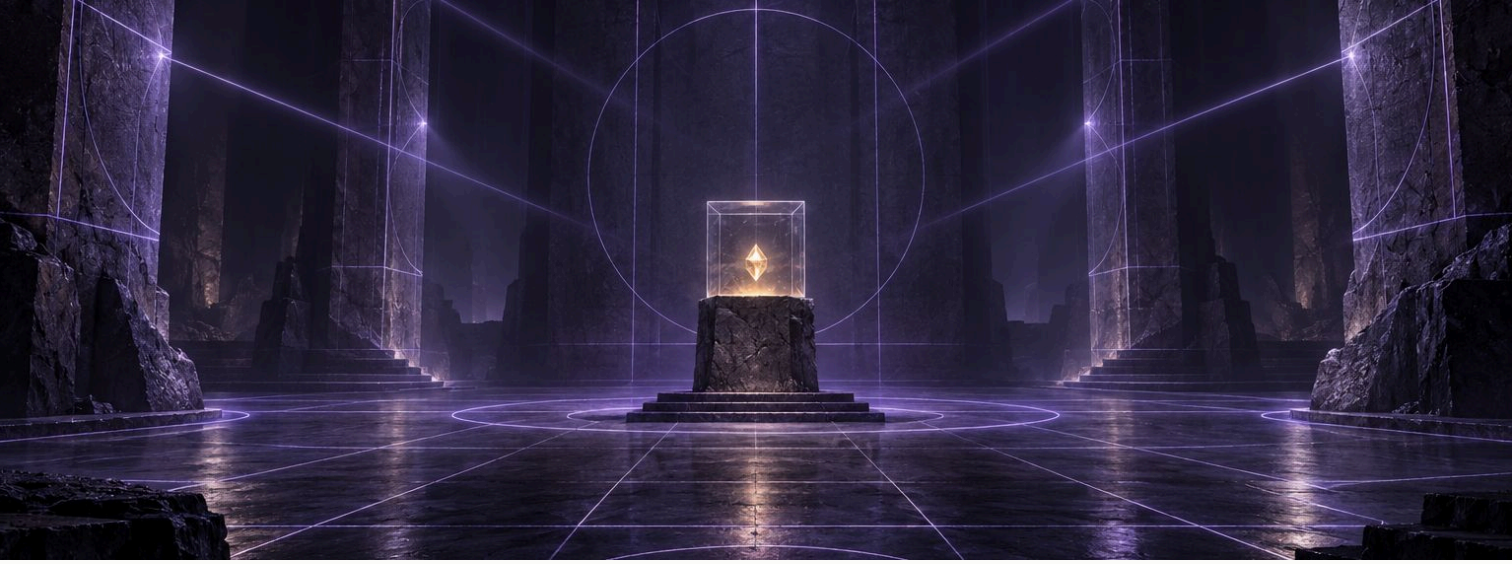
Η ομάδα σου τρέχει τα plans και τα applies σε HCP Terraform με VCS workflow και δεν έχει πρόσβαση στο state από τους υπολογιστές της. Ένα workspace πρέπει να πάψει να διαχειρίζεται ένα S3 bucket, χωρίς να το σβήσει. Τι κάνεις;

a. Σβήνεις το resource block και κάνεις merge.
b. Τρέχεις `terraform state rm` στον υπολογιστή σου.
c. Προσθέτεις `prevent_destroy = true` στο `lifecycle` του resource.
d. Αντικαθιστάς το resource block με block `removed` και `lifecycle { destroy = false }`.

ΕΡΩΤΗΣΗ 23.6

Ένα resource έχει `count = 2`. Το αλλάζεις σε `for_each` με τα κλειδιά `"logs"` και `"backups"`. Πώς αποφεύγεις την αντικατάσταση των δύο buckets;

a. Με ένα block `moved` για κάθε instance, από `[0]` και `[1]` στο αντίστοιχο κλειδί.
b. Με ένα block `moved` από `aws_s3_bucket.b` σε `aws_s3_bucket.b`.
c. Δεν χρειάζεται τίποτα. Το Terraform αντιστοιχίζει μόνο του θέσεις σε κλειδιά.
d. Με `terraform apply -replace` για κάθε instance.



ΚΕΦΑΛΑΙΟ 24

Secrets, sensitive, ephemeral και Vault

Κάθε υποδομή έχει μυστικά. Κωδικούς βάσεων, API keys και tokens. Σε αυτό το κεφάλαιο βλέπεις πού μπορεί να καταλήξει ένα μυστικό όταν περνά από το Terraform και πώς το κρατάς μακριά από την οθόνη, από το state και από το Git. Τελειώνεις με το Vault, το εργαλείο της HashiCorp που φυλά τα μυστικά και τα δίνει μόνο σε όποιον δικαιούται.

Πού διαρρέει ένα secret

Ένα **secret** είναι μια τιμή που δεν πρέπει να δει κανείς άλλος, όπως ένας κωδικός, ένα κλειδί ή ένα token. Σε ένα τυπικό run του Terraform ένα secret μπορεί να μείνει σε τέσσερα σημεία.

1. **Ο κώδικας.** Μια τιμή γραμμένη σε `.tf` ή σε `.tfvars` που μπαίνει σε Git τη βλέπει όποιος έχει πρόσβαση στο repository. Το κεφάλαιο 7 έδειξε πώς την κρατάς έξω.
2. **Η οθόνη.** Το plan και το apply τυπώνουν τιμές. Στο CI η έξοδος μένει στα logs, που τα διαβάζουν πολλοί.
3. **Το state.** Το Terraform γράφει εκεί κάθε attribute σε απλό κείμενο, όπως είδες στο κεφάλαιο 4.
4. **Το plan file.** Το αρχείο του `plan -out` του κεφαλαίου 3 κρατά τις τιμές των μεταβλητών και όσα θα γράψει το apply.

Για κάθε σημείο υπάρχει διαφορετικό εργαλείο. Το `sensitive` κρύβει τιμές από την οθόνη. Οι **ephemeral** τιμές και τα **write-only arguments** δεν γράφονται ποτέ στο state ή στο plan file. Ένας secret manager όπως το Vault κρατά την τιμή έξω από τον κώδικα. Θα τα δεις με αυτή τη σειρά.

sensitive σε μεταβλητές

Μια εφαρμογή θέλει ένα αρχείο ρυθμίσεων με το connection string της βάσης της. Το string περιέχει τον κωδικό. Φτιάξε φάκελο `sens` με δύο αρχεία.

TERMINAL

```
mkdir sens
cd sens
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

locals {
  dsn = "postgres://app:${var.db_password}@db.internal:5432/shop"
}

resource "local_file" "env" {
  filename = "app.env"
  content  = "DATABASE_URL=${local.dsn}\n"
}
```

variables.tf

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

Η μεταβλητή έχει `sensitive = true`. Δεν έχει `default`, γιατί ένας κωδικός δεν γράφεται σε κώδικα. Την τιμή τη δίνεις με μεταβλητή περιβάλλοντος `TF_VAR_`, όπως στο κεφάλαιο 7, ώστε να μη μείνει σε κανένα αρχείο του project.

TERMINAL

```
export TF_VAR_db_password='Pagoto-Fraoula-42'
terraform init
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
```

```
  + create

Terraform will perform the following actions:

  # local_file.env will be created
  + resource "local_file" "env" {
      + content              = (sensitive value)
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "app.env"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

──────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run "terraform apply" now.
```

Στη θέση του `content` το plan γράφει `(sensitive value)`. Πρόσεξε ότι το `content` δεν είναι η μεταβλητή. Είναι ένα string που φτιάχτηκε από το `local.dsn`, που με τη σειρά του φτιάχτηκε από τη μεταβλητή. Η ιδιότητα sensitive **διαδίδεται**. Κάθε έκφραση που χρησιμοποιεί μια sensitive τιμή δίνει sensitive αποτέλεσμα. Το βλέπεις και στο `terraform console` του κεφαλαίου 9.

TERRAFORM CONSOLE

```
> var.db_password
(sensitive value)
> local.dsn
(sensitive value)
> upper(var.db_password)
(sensitive value)
> length(var.db_password)
(sensitive value)
```

Ακόμη και το μήκος του κωδικού κρύβεται. Το Terraform δεν ξέρει ποιο κομμάτι μιας έκφρασης είναι επικίνδυνο, οπότε σημαδεύει όλο το αποτέλεσμα.

sensitive σε outputs

Πρόσθεσε ένα output με το connection string, για να το διαβάζει ένα script.

outputs.tf

```
output "dsn" {
  value = local.dsn
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Output refers to sensitive values
│
│   on outputs.tf line 1:
│    1: output "dsn" {
│
│ To reduce the risk of accidentally exporting sensitive data that was
│ intended to be only internal, Terraform requires that any root module
│ output containing sensitive data be explicitly marked as sensitive, to
│ confirm your intent.
│
│ If you do intend to export this data, annotate the output value as
│ sensitive by adding the following argument:
│     sensitive = true
╵
```

Το Terraform σταματά με `Output refers to sensitive values`. Τα outputs του root module τυπώνονται στο τέλος κάθε apply και τα διαβάζουν άλλα εργαλεία. Αν ένα output περιέχει sensitive τιμή, το Terraform θέλει να το δηλώσεις ρητά, ώστε η διαρροή να μη γίνει κατά λάθος. Το μήνυμα λέει και τη λύση.

outputs.tf

```
output "dsn" {
  value     = local.dsn
  sensitive = true
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.env will be created
  + resource "local_file" "env" {
      + content              = (sensitive value)
      …
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "app.env"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + dsn = (sensitive value)
local_file.env: Creating...
local_file.env: Creation complete after 0s [id=c79abbbfe9081ccd8eed82572b0dcb514cfb31e3]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

```
Outputs:

dsn = <sensitive>
```

Το plan γράφει `(sensitive value)` στο `content` και στο output. Στο τέλος του apply το output εμφανίζεται ως `<sensitive>`. Η τιμή όμως δεν χάθηκε.

TERMINAL

```
terraform output
terraform output dsn
terraform output -raw dsn
```

ΕΞΟΔΟΣ

```
dsn = <sensitive>
"postgres://app:Pagoto-Fraoula-42@db.internal:5432/shop"
postgres://app:Pagoto-Fraoula-42@db.internal:5432/shop
```

Η λίστα κρύβει την τιμή. Όταν ζητάς ένα συγκεκριμένο output με το όνομά του, το Terraform θεωρεί ότι το θέλεις και το τυπώνει, σε εισαγωγικά ή με `-raw` χωρίς αυτά. Το ίδιο κάνει και το `-json`.

TERMINAL

```
terraform output -json
```

ΕΞΟΔΟΣ

```
{
  "dsn": {
    "sensitive": true,
    "type": "string",
    "value": "postgres://app:Pagoto-Fraoula-42@db.internal:5432/shop"
  }
}
```

Το πεδίο `"sensitive": true` λέει στο πρόγραμμα που διαβάζει το JSON ότι η τιμή θέλει προσοχή. Την τιμή όμως τη δίνει ολόκληρη.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Το `sensitive = true` κρύβει την τιμή στην έξοδο του plan και του apply και στη λίστα του `terraform output`. Το `terraform output ΟΝΟΜΑ`, το `-raw` και το `-json` τυπώνουν την πραγματική τιμή. Ένα output του root module που περιέχει sensitive τιμή χωρίς `sensitive = true` είναι σφάλμα, όχι προειδοποίηση.

Η συνάρτηση nonsensitive

Μερικές φορές μια τιμή που προέρχεται από sensitive τιμή δεν είναι μυστική. Το μήκος του κωδικού, για παράδειγμα, χρειάζεται σε έναν έλεγχο πολιτικής. Η συνάρτηση `nonsensitive` αφαιρεί το σημάδι.

TERRAFORM CONSOLE

```
> nonsensitive(length(var.db_password))
17
> nonsensitive(var.db_password)
"Pagoto-Fraoula-42"
```

Η πρώτη χρήση είναι λογική. Η δεύτερη τυπώνει τον κωδικό στην οθόνη. Η `nonsensitive` δεν ελέγχει τι αποκαλύπτει. Κάνει ό,τι της ζητήσεις.

> **ΠΑΓΙΔΑ**
> Μη βάζεις `nonsensitive` μόνο και μόνο για να φύγει ένα σφάλμα. Πρώτα ρώτα αν η τιμή είναι πράγματι ακίνδυνη. Αν δεν είναι, η σωστή διόρθωση είναι `sensitive = true` στο output.

Τι μένει στο state και στο plan file

Το `terraform state show` κρύβει κι αυτό τις sensitive τιμές.

TERMINAL

```
terraform state show local_file.env
```

ΕΞΟΔΟΣ

```
…
    content              = (sensitive value)
…
```

Το αρχείο του state όμως έχει την τιμή. Ψάξ' την με `jq`.

TERMINAL

```
jq -r '.resources[0].instances[0].attributes.content' terraform.tfstate
jq '.outputs.dsn' terraform.tfstate
jq -c '.resources[0].instances[0].sensitive_attributes' terraform.tfstate
```

ΕΞΟΔΟΣ

```
DATABASE_URL=postgres://app:Pagoto-Fraoula-42@db.internal:5432/shop

{
  "value": "postgres://app:Pagoto-Fraoula-42@db.internal:5432/shop",
  "type": "string",
  "sensitive": true
}
[[{"type":"get_attr","value":"content"}],[{"type":"get_attr","value":"sensitive_content"}]]
```

Ο κωδικός είναι σε απλό κείμενο, στο attribute του resource και στο output. Το πεδίο `sensitive_attributes` κρατά μόνο τη λίστα με τα attributes που πρέπει να κρύβονται στην οθόνη. Εδώ είναι το `content`, που σημαδεύτηκε από τη μεταβλητή. Είναι επίσης το `sensitive_content`, που ο provider δηλώνει πάντα sensitive. Το state δεν κρυπτογραφεί τίποτα.

Το ίδιο ισχύει για το plan file. Φτιάξε ένα και διάβασέ το με `terraform show -json`.

TERMINAL

```
terraform plan -out=app.tfplan
```

TERMINAL

```
terraform show -json app.tfplan | jq '.variables'
```

ΕΞΟΔΟΣ

```
{
  "db_password": {
    "value": "Pagoto-Fraoula-42"
  }
}
```

Το plan file κρατά την τιμή κάθε μεταβλητής, ώστε το apply του αρχείου να δουλέψει χωρίς να του τη δώσεις ξανά. Όποιος έχει το αρχείο έχει και τον κωδικό.

Πώς προστατεύεις το state

Αφού το state και το plan file περιέχουν μυστικά, τα φυλάς όπως τα ίδια τα μυστικά.

- Δεν μπαίνουν σε Git. Το `.gitignore` του κεφαλαίου 4 κρατά έξω τα `*.tfstate`. Πρόσθεσε και το `*.tfplan`.
- Το state ζει σε remote backend με κρυπτογράφηση. Στο S3 backend του κεφαλαίου 19 το argument `encrypt = true` ζητά ρητά κρυπτογράφηση του αρχείου στο S3. Με `kms_key_id` η κρυπτογράφηση γίνεται με δικό σου κλειδί του KMS, οπότε για να διαβάσει κανείς το state χρειάζεται άδεια και για το κλειδί.
- Η πρόσβαση στο bucket περιορίζεται με IAM policies στους ανθρώπους και στα συστήματα που τρέχουν πραγματικά το Terraform. Όποιος διαβάζει το state διαβάζει και τα μυστικά του.
- Το HCP Terraform του κεφαλαίου 26 κρυπτογραφεί το state και ελέγχει ποιος το διαβάζει με δικαιώματα ανά ομάδα.
- Τα plan files που κρατά ένα σύστημα CI είναι κι αυτά μυστικά. Τα σβήνεις μόλις γίνει το apply.

HCL

```
terraform {
  backend "s3" {
    bucket       = "tfbook-state"
    key          = "shop/terraform.tfstate"
    region       = "eu-central-1"
    encrypt      = true
    use_lockfile = true
  }
}
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το state και το plan file κρατούν τις sensitive τιμές σε απλό κείμενο. Το `sensitive` δεν είναι κρυπτογράφηση. Η σωστή προστασία είναι remote backend με κρυπτογράφηση, αυστηρός έλεγχος πρόσβασης και ποτέ Git.



Ephemeral τιμές

Το `sensitive` λύνει το πρόβλημα της οθόνης. Για το state και το plan file το Terraform 1.10 έφερε τις **ephemeral** τιμές. Μια ephemeral τιμή υπάρχει μόνο όσο τρέχει μια εντολή. Το Terraform δεν τη γράφει ποτέ στο state ούτε στο plan file. Στο επόμενο run την ξαναπαίρνει από την πηγή της.

Το τίμημα είναι ότι μια ephemeral τιμή δεν πηγαίνει οπουδήποτε. Ένα κανονικό argument ενός resource καταλήγει στο state, άρα δεν τη δέχεται. Επιτρέπεται μόνο σε θέσεις που δεν αποθηκεύονται.

- Σε **write-only arguments** των resources, που θα δεις παρακάτω.
- Στο block `provider`, για παράδειγμα ως κωδικός ή token του provider.
- Στα arguments άλλων ephemeral resources.
- Σε locals, που γίνονται κι αυτά ephemeral.
- Σε ephemeral μεταβλητές και ephemeral outputs ενός child module.
- Σε blocks `provisioner` και `connection`. Ένα provisioner είδες στο κεφάλαιο 14.

Ephemeral μεταβλητές

Μια μεταβλητή γίνεται ephemeral με `ephemeral = true`. Δοκίμασέ το στη μεταβλητή του `sens`.

variables.tf

```
variable "db_password" {
  type      = string
  sensitive = true
  ephemeral = true
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid use of ephemeral value
│
│   with local_file.env,
│   on main.tf line 16, in resource "local_file" "env":
│   16:   content  = "DATABASE_URL=${local.dsn}\n"
│
│ Ephemeral values are not valid for "content", because it is not a
│ write-only attribute and must be persisted to state.
╵

╷
│ Error: Ephemeral value not allowed
│
│   on outputs.tf line 2, in output "dsn":
│    2:   value     = local.dsn
│
│ This output value is not declared as returning an ephemeral value, so it
│ cannot be set to a result derived from an ephemeral value.
╵
```

Δύο σφάλματα. Το πρώτο, `Invalid use of ephemeral value`, αφορά το `content` του `local_file`. Το μήνυμα εξηγεί ότι το `content` δεν είναι write-only attribute και πρέπει να αποθηκευτεί στο state. Το `content` δεν αναφέρει τη μεταβλητή απευθείας αλλά το `local.dsn`. Η ιδιότητα ephemeral διαδίδεται όπως η sensitive. Το δεύτερο, `Ephemeral value not allowed`, αφορά το output. Τα outputs του root module γράφονται στο state. Ούτε ένα `ephemeral = true` στο output θα βοηθούσε, γιατί τα ephemeral outputs επιτρέπονται μόνο σε child modules.

Αυτή η configuration χρειάζεται τον κωδικό μέσα στο αρχείο `app.env`, οπότε η μεταβλητή μένει απλώς sensitive. Επανάφερε το `variables.tf`.

variables.tf

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

Μια ephemeral μεταβλητή συμπεριφέρεται διαφορετικά και με αποθηκευμένο plan. Αφού δεν γράφεται στο plan file, πρέπει να τη δώσεις ξανά όταν τρέχεις `terraform apply` με το αρχείο. Αλλιώς το Terraform σταματά με το σφάλμα `No value for required variable`.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Μια ephemeral τιμή χρειάζεται προορισμό που δεν αποθηκεύεται. Αν το resource που σε ενδιαφέρει δεν έχει write-only argument, η τιμή θα καταλήξει στο state. Τότε σου μένουν το `sensitive` και η προστασία του state.

Ephemeral resources

Ένα **ephemeral resource** γράφεται με block `ephemeral` αντί για `resource`. Διαβάζει ή παράγει μια τιμή σε κάθε run και την ξεχνά στο τέλος. Ο provider `random` 3.9.1 έχει ephemeral εκδοχή του `random_password`. Βάλε τις δύο εκδοχές δίπλα δίπλα σε νέο φάκελο.

TERMINAL

```
cd ..
mkdir ephem
cd ephem
```

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_password" "stored" {
  length = 16
}

ephemeral "random_password" "temp" {
  length = 16
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
ephemeral.random_password.temp: Opening...
ephemeral.random_password.temp: Opening complete after 0s
ephemeral.random_password.temp: Closing...
ephemeral.random_password.temp: Closing complete after 0s

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # random_password.stored will be created
  + resource "random_password" "stored" {
      + bcrypt_hash = (sensitive value)
      + id          = (known after apply)
      + length      = 16
      + lower       = true
      + min_lower   = 0
```

```
      + min_numeric = 0
      + min_special = 0
      + min_upper   = 0
      + number      = true
      + numeric     = true
      + result      = (sensitive value)
      + special     = true
      + upper       = true
    }

Plan: 1 to add, 0 to change, 0 to destroy.
ephemeral.random_password.temp: Opening...
random_password.stored: Creating...
ephemeral.random_password.temp: Opening complete after 0s
ephemeral.random_password.temp: Closing...
random_password.stored: Creation complete after 0s [id=none]
ephemeral.random_password.temp: Closing complete after 0s

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το plan αναφέρει μόνο το `random_password.stored`, με `result = (sensitive value)`. Το ephemeral resource δεν μπαίνει καν στο plan. Γι' αυτό εμφανίζονται μόνο οι γραμμές `Opening` και `Closing`. Το Terraform το «άνοιξε» για να πάρει την τιμή και το «έκλεισε» όταν δεν τη χρειαζόταν πια. Αυτό έγινε δύο φορές, μία στο plan και μία στο apply. Κάθε άνοιγμα δίνει νέο κωδικό.

TERMINAL

```
terraform state list
jq -r '.resources[].instances[0].attributes.result' terraform.tfstate
```

ΕΞΟΔΟΣ

```
random_password.stored
0<-FYNU<hA<4W[=-
```

Στο state υπάρχει μόνο το `random_password.stored` και ο κωδικός του είναι εκεί σε απλό κείμενο. Το ephemeral δεν άφησε ίχνος. Σε ένα ephemeral resource αναφέρεσαι με το πρόθεμα `ephemeral.`, για παράδειγμα `ephemeral.random_password.temp.result`. Δοκίμασε να γράψεις τον κωδικό του σε αρχείο.

app.tf

```
resource "local_file" "env" {
  filename = "app.env"
  content  = "DB_PASSWORD=${ephemeral.random_password.temp.result}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Invalid use of ephemeral value
│
│   with local_file.env,
│   on app.tf line 3, in resource "local_file" "env":
│    3:   content  = "DB_PASSWORD=${ephemeral.random_password.temp.result}\n"
│
│ Ephemeral values are not valid for "content", because it is not a
│ write-only attribute and must be persisted to state.
╵
```

Είναι το ίδιο σφάλμα με της ephemeral μεταβλητής. Το `local_file` δεν έχει write-only argument, άρα η τιμή δεν έχει πού να πάει. Σβήσε το `app.tf`.

TERMINAL

```
rm app.tf
```

Write-only arguments

Ένα **write-only argument** δέχεται τιμή στο plan και στο apply, τη στέλνει στο API και δεν την αποθηκεύει πουθενά. Υπάρχουν από το Terraform 1.11 και τα ονόματά τους τελειώνουν συνήθως σε `_wo`. Τα δηλώνει ο provider. Δεν μετατρέπεις εσύ ένα κανονικό argument σε write-only.

Ο AWS provider έχει το `secret_string_wo` στο resource `aws_secretsmanager_secret_version`. Ο AWS Secrets Manager κρατά secrets με όνομα και εκδόσεις. Η εφαρμογή διαβάζει τον κωδικό από εκεί όταν ξεκινά, οπότε δεν χρειάζεται αρχείο ρυθμίσεων με τον κωδικό. Φτιάξε φάκελο `aws`. Το Moto τρέχει όπως στο κεφάλαιο 17 και οι μεταβλητές `AWS_` είναι ορισμένες.

TERMINAL

```
cd ..
mkdir aws
cd aws
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

resource "aws_secretsmanager_secret" "db" {
  name = "shop/db-password"
}
```

password.tf

```
ephemeral "random_password" "db" {
  length  = 24
  special = false
}

resource "aws_secretsmanager_secret_version" "db" {
  secret_id               = aws_secretsmanager_secret.db.id
  secret_string_wo        = ephemeral.random_password.db.result
  secret_string_wo_version = 1
}
```

Το `aws_secretsmanager_secret` φτιάχνει το secret, δηλαδή το όνομα. Το `aws_secretsmanager_secret_version` γράφει μέσα του μια τιμή.

- Το `secret_string_wo` παίρνει τον ephemeral κωδικό. Είναι write-only, οπότε δέχεται ephemeral τιμή.
- Το `secret_string_wo_version` είναι κανονικό argument, ένας αριθμός που γράφεται στο state. Το Terraform δεν θυμάται την τιμή, άρα δεν μπορεί να καταλάβει μόνο του πότε άλλαξε. Το version είναι ο τρόπος να του το πεις.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
ephemeral.random_password.db: Opening...
ephemeral.random_password.db: Opening complete after 0s
ephemeral.random_password.db: Closing...
ephemeral.random_password.db: Closing complete after 0s

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_secretsmanager_secret.db will be created
  + resource "aws_secretsmanager_secret" "db" {
      + arn                            = (known after apply)
      + force_overwrite_replica_secret = false
      + id                             = (known after apply)
      + name                           = "shop/db-password"
      + name_prefix                    = (known after apply)
      + policy                         = (known after apply)
      + recovery_window_in_days        = 30
      + region                         = "eu-central-1"
      + tags_all                       = (known after apply)
    }

  # aws_secretsmanager_secret_version.db will be created
  + resource "aws_secretsmanager_secret_version" "db" {
      + arn                      = (known after apply)
      + has_secret_string_wo     = (known after apply)
      + id                       = (known after apply)
      + region                   = "eu-central-1"
      + secret_arn               = (known after apply)
      + secret_id                = (known after apply)
      + secret_string_wo         = (write-only attribute)
      + secret_string_wo_version = 1
      + version_id               = (known after apply)
      + version_stages           = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.
ephemeral.random_password.db: Opening...
aws_secretsmanager_secret.db: Creating...
ephemeral.random_password.db: Opening complete after 0s
…
aws_secretsmanager_secret_version.db: Creating...
…
ephemeral.random_password.db: Closing...
ephemeral.random_password.db: Closing complete after 0s

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

Το plan δείχνει `secret_string_wo = (write-only attribute)`. Δεν γράφει καν `(sensitive value)`, γιατί δεν υπάρχει τιμή να κρυφτεί. Το `has_secret_string_wo` θα πει μετά το apply μόνο αν δόθηκε τιμή, όχι ποια. Το ephemeral `random_password.db` άνοιξε στο plan και ξανά στο apply, όπως στο `ephem`. Οι γραμμές που λείπουν έχουν μακριά ARNs και δεν προσθέτουν τίποτα.

TERMINAL

```
jq '.resources[]
  | select(.type == "aws_secretsmanager_secret_version")
  | .instances[0].attributes
  | {secret_string, secret_string_wo, secret_string_wo_version, has_secret_string_wo}' \
  terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "secret_string": "",
  "secret_string_wo": null,
  "secret_string_wo_version": 1,
  "has_secret_string_wo": true
}
```

Το `secret_string_wo` είναι `null` στο state. Το `secret_string` είναι το κανονικό argument που δεν χρησιμοποιήσαμε, γι' αυτό είναι κενό. Ο κωδικός δεν υπάρχει πουθενά στο αρχείο. Υπάρχει όμως στο Secrets Manager. Ρώτησέ το με το `aws` CLI.

TERMINAL

```
aws secretsmanager get-secret-value --secret-id shop/db-password \
  --query SecretString --output text
```

ΕΞΟΔΟΣ

```
G59iSWErJkwdgE1OxdMYIE6I
```

Αυτός είναι ο κωδικός που έφτιαξε το ephemeral resource στο apply. Τρέξε τώρα ένα plan χωρίς καμία αλλαγή.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
ephemeral.random_password.db: Opening...
ephemeral.random_password.db: Opening complete after 0s
…
ephemeral.random_password.db: Closing...
ephemeral.random_password.db: Closing complete after 0s

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Το ephemeral resource έδωσε νέο κωδικό και σε αυτό το plan. Το Terraform όμως δεν βλέπει διαφορά, γιατί το version έμεινε `1`. Ο Secrets Manager κρατά τον πρώτο κωδικό. Για να γραφτεί νέος, για παράδειγμα σε μια τακτική αλλαγή κωδικών, αύξησε το version.

password.tf

```
ephemeral "random_password" "db" {
  length  = 24
  special = false
}

resource "aws_secretsmanager_secret_version" "db" {
  secret_id                = aws_secretsmanager_secret.db.id
  secret_string_wo         = ephemeral.random_password.db.result
  secret_string_wo_version = 2
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
ephemeral.random_password.db: Opening...
ephemeral.random_password.db: Opening complete after 0s
…
ephemeral.random_password.db: Closing...
ephemeral.random_password.db: Closing complete after 0s

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # aws_secretsmanager_secret_version.db must be replaced
-/+ resource "aws_secretsmanager_secret_version" "db" {
      …
      ~ has_secret_string_wo     = true -> (known after apply)
      …
      ~ secret_string_wo_version = 1 -> 2 # forces replacement
      ~ version_id               = "terraform-hUaBLERQxIIF0TkaBP6yiTxx6K" -> (known after apply)
      ~ version_stages           = [
          - "AWSCURRENT",
        ] -> (known after apply)
        # (5 unchanged attributes hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.
ephemeral.random_password.db: Opening...
ephemeral.random_password.db: Opening complete after 0s
…
aws_secretsmanager_secret_version.db: Destruction complete after 0s
aws_secretsmanager_secret_version.db: Creating...
…
ephemeral.random_password.db: Closing...
ephemeral.random_password.db: Closing complete after 0s

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Η αλλαγή `1 -> 2` έχει δίπλα της `# forces replacement`, οπότε το Terraform αντικαθιστά το resource με `-/+`. Στο Secrets Manager αυτό σημαίνει νέα έκδοση του secret με τον νέο κωδικό.

TERMINAL

```
aws secretsmanager get-secret-value --secret-id shop/db-password \
  --query SecretString --output text
aws secretsmanager list-secret-version-ids --secret-id shop/db-password \
  --query 'Versions[].VersionStages[]' --output text
```

ΕΞΟΔΟΣ

```
WgDy99slW5hmWktMUxLRO6VB
AWSPREVIOUS AWSCURRENT
```

Ο κωδικός είναι διαφορετικός από τον πρώτο. Το secret έχει πλέον δύο εκδόσεις. Η νέα έχει την ετικέτα `AWSCURRENT` και την επιστρέφει το `get-secret-value`. Η παλιά κρατά την ετικέτα `AWSPREVIOUS`.

ΠΑΓΙΔΑ

Αν αλλάξεις μόνο την τιμή που δίνεις σε ένα write-only argument, το plan λέει `No changes`. Το Terraform δεν έχει την παλιά τιμή για να συγκρίνει. Κάθε φορά που θέλεις να γραφτεί νέα τιμή, αυξάνεις το version.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Στο πραγματικό AWS κάθε secret του Secrets Manager κοστίζει περίπου 0,40 δολάρια τον μήνα. Το `destroy` δεν το σβήνει αμέσως. Το προγραμματίζει για διαγραφή μετά από 30 ημέρες, όπως λέει το `recovery_window_in_days` του plan. Ως τότε δεν μπορείς να φτιάξεις άλλο secret με το ίδιο όνομα. Σε δοκιμές βάλε `recovery_window_in_days = 0` για άμεση διαγραφή.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Τα ephemeral resources και οι ephemeral μεταβλητές ήρθαν στο Terraform 1.10 και τα write-only arguments στο 1.11. Υπάρχουν λοιπόν στην 1.12, στην οποία βασίζεται η εξέταση. Ο κανόνας είναι απλός. Το `sensitive` κρύβει από την οθόνη. Το ephemeral και το write-only κρατούν την τιμή έξω από το state και το plan file.

Το Vault

Τα μυστικά πρέπει να ζουν κάπου. Το **Vault** της HashiCorp είναι ένας κεντρικός secret manager. Κρατά τα secrets κρυπτογραφημένα, δίνει πρόσβαση με tokens και policies και μπορεί να καταγράφει ποιος διάβασε τι. Τα secrets οργανώνονται σε **secrets engines**. Κάθε engine ζει σε ένα path, που



λέγεται mount. Το πιο απλό engine είναι το KV, που κρατά ζεύγη κλειδιού και τιμής. Η δεύτερη έκδοσή του, το KV v2, κρατά και ιστορικό εκδόσεων για κάθε secret.

Για τις δοκιμές τρέχει ένας Vault **dev server**. Ξεκινά έτοιμος, κρατά τα πάντα στη μνήμη και έχει ένα KV v2 στο mount `secret`. Το root token του το ορίζει η μεταβλητή `VAULT_DEV_ROOT_TOKEN_ID`, εδώ `root`. Χωρίς αυτήν ο server φτιάχνει τυχαίο token. Τον σηκώνεις με Docker και ορίζεις δύο μεταβλητές περιβάλλοντος.

TERMINAL

```
docker run -d --name vault --cap-add IPC_LOCK \
  -e VAULT_DEV_ROOT_TOKEN_ID=root \
  -p 127.0.0.1:8200:8200 hashicorp/vault:1.21.4
export VAULT_ADDR=http://127.0.0.1:8200
export VAULT_TOKEN=root
```

ΠΑΓΙΔΑ

Ο dev server είναι μόνο για δοκιμές. Χάνει τα πάντα όταν σταματήσει και το root token του επιτρέπει τα πάντα. Ένας πραγματικός Vault έχει μόνιμη αποθήκευση και TLS. Δίνει σε κάθε σύστημα token με όσα δικαιώματα χρειάζεται και τίποτα παραπάνω.

Το `VAULT_ADDR` λέει πού είναι ο server και το `VAULT_TOKEN` με ποιο token μιλάς. Τα διαβάζει και ο provider του Terraform. Δεν χρειάζεσαι το CLI του Vault. Αρκεί το HTTP API με `curl`. Κάθε κλήση που θέλει άδεια στέλνει το token στο header `X-Vault-Token`.

TERMINAL

```
echo $VAULT_ADDR
curl -s $VAULT_ADDR/v1/sys/health | jq '{initialized, sealed, version}'
```

ΕΞΟΔΟΣ

```
http://127.0.0.1:8200
{
  "initialized": true,
  "sealed": false,
  "version": "1.21.4"
}
```

Ο server είναι έτοιμος και ανοιχτός (`sealed` είναι `false`). Η ομάδα ασφαλείας γράφει εκεί τα στοιχεία της βάσης. Το Terraform δεν τα φτιάχνει. Μόνο τα διαβάζει.

TERMINAL

```
curl -s -H "X-Vault-Token: $VAULT_TOKEN" -X POST \
  -d '{"data": {"username": "shop", "password": "Melomakarono-77"}}' \
  $VAULT_ADDR/v1/secret/data/shop/db | jq '.data.version'
```

ΕΞΟΔΟΣ

```
1
```

Το KV v2 αποθήκευσε την πρώτη έκδοση του secret `shop/db`. Στο API το path έχει το `data` ανάμεσα στο mount και στο όνομα. Στον provider δίνεις χωριστά το mount και το όνομα.

Ο provider και ένα data source

TERMINAL

```
cd ..
mkdir vault
cd vault
```

main.tf

```
terraform {
  required_providers {
    vault = {
      source  = "hashicorp/vault"
      version = "5.12.0"
    }
  }
}

provider "vault" {}
```

Το block `provider "vault"` είναι άδειο. Ο provider παίρνει διεύθυνση και token από το `VAULT_ADDR` και το `VAULT_TOKEN`, οπότε τίποτα από τα δύο δεν γράφεται σε κώδικα. Αν προτιμάς να δίνεις το token ως μεταβλητή, κάν' τη ephemeral. Το block `provider` είναι από τις θέσεις που δέχονται ephemeral τιμές.

HCL

```
variable "vault_token" {
  type      = string
  ephemeral = true
}

provider "vault" {
  token = var.vault_token
}
```

Τώρα διάβασε το secret με data source, όπως έμαθες στο κεφάλαιο 12.

read.tf

```
data "vault_kv_secret_v2" "db" {
  mount = "secret"
  name  = "shop/db"
}

output "db_user" {
  value = nonsensitive(data.vault_kv_secret_v2.db.data["username"])
}
```

Το attribute `data` είναι map με τα κλειδιά του secret και ο provider το δηλώνει sensitive. Το username όμως δεν είναι μυστικό, οπότε το output το βγάζει με `nonsensitive`. Αυτή είναι η σωστή χρήση της συνάρτησης.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
data.vault_kv_secret_v2.db: Reading...
data.vault_kv_secret_v2.db: Read complete after 0s [id=secret/data/shop/db]

Changes to Outputs:
  + db_user = "shop"

You can apply this plan to save these new output values to the Terraform
state, without changing any real infrastructure.

│ Warning: Deprecated Resource
│
│   with data.vault_kv_secret_v2.db,
│   on read.tf line 1, in data "vault_kv_secret_v2" "db":
│    1: data "vault_kv_secret_v2" "db" {
│
│ Deprecated. Please use new Ephemeral KVV2 Secret resource
│ `vault_kv_secret_v2` instead
│
│ (and one more similar warning elsewhere)

│ Warning: Deprecated value used
│
│   on read.tf line 7, in output "db_user":
│    7:   value = nonsensitive(data.vault_kv_secret_v2.db.data["username"])
│
│   The deprecation originates from data.vault_kv_secret_v2.db
│
│ Deprecated. Please use new Ephemeral KVV2 Secret resource
│ `vault_kv_secret_v2` instead
│
│ (and one more similar warning elsewhere)

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

Outputs:

db_user = "shop"
```

Το output βγήκε σωστό, αλλά ο provider έβγαλε προειδοποιήσεις. Η `Deprecated Resource` λέει ότι το data source `vault_kv_secret_v2` είναι deprecated στην 5.12.0 και ότι πρέπει να χρησιμοποιείς το ephemeral resource με το ίδιο όνομα. Η `Deprecated value used` λέει το ίδιο για το output που χρησιμοποιεί την τιμή του. Το Terraform μαζεύει όμοιες προειδοποιήσεις σε μία και γράφει πόσες ακόμη υπάρχουν, εδώ με τη γραμμή `and one more similar warning elsewhere`. Ο λόγος φαίνεται στο state.

TERMINAL

```
jq '.resources[] | select(.mode == "data") | .instances[0].attributes.data' \
  terraform.tfstate
```

ΕΞΟΔΟΣ

```
{
  "password": "Melomakarono-77",
  "username": "shop"
}
```

Ο κωδικός είναι εκεί σε απλό κείμενο. Ό,τι διαβάζει ένα data source αποθηκεύεται στο state, όπως κάθε attribute. Το Vault φύλαγε τον κωδικό κρυπτογραφημένο και τον έδινε μόνο με token. Το Terraform τον αντέγραψε σε ένα αρχείο που προστατεύεται μόνο όσο προστατεύεις το state.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Όταν το Terraform διαβάζει ένα secret από το Vault με data source, η τιμή γράφεται στο state και στο plan file. Το Vault δεν προστατεύει ό,τι έχει ήδη αντιγραφεί έξω από αυτό. Η χρήση του Vault δεν σε απαλλάσσει από την προστασία του state.

Από το Vault χωρίς state

Το ephemeral resource `vault_kv_secret_v2` διαβάζει το ίδιο secret χωρίς να το αποθηκεύσει. Ας πούμε ότι η εφαρμογή θέλει τον κωδικό στο δικό της secret, `shop/app`. Το resource `vault_kv_secret_v2` γράφει secrets στο Vault και έχει το write-only argument `data_json_wo`. Αντικατάστησε το `read.tf` με το `copy.tf`.

TERMINAL

```
rm read.tf
```

copy.tf

```
ephemeral "vault_kv_secret_v2" "db" {
  mount = "secret"
  name  = "shop/db"
}

resource "vault_kv_secret_v2" "app" {
  mount = "secret"
  name  = "shop/app"
  data_json_wo = jsonencode({
    DATABASE_PASSWORD = ephemeral.vault_kv_secret_v2.db.data.password
  })
  data_json_wo_version = 1
}
```

Το `data_json_wo` θέλει JSON, οπότε η `jsonencode` του κεφαλαίου 10 φτιάχνει το object. Το `data_json_wo_version` κάνει τη δουλειά του `secret_string_wo_version` που είδες στο AWS.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
ephemeral.vault_kv_secret_v2.db: Opening...
ephemeral.vault_kv_secret_v2.db: Opening complete after 0s
ephemeral.vault_kv_secret_v2.db: Closing...
ephemeral.vault_kv_secret_v2.db: Closing complete after 0s

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # vault_kv_secret_v2.app will be created
  + resource "vault_kv_secret_v2" "app" {
      + data                 = (sensitive value)
      + data_json_wo         = (write-only attribute)
      + data_json_wo_version = 1
      + delete_all_versions  = false
      + disable_read         = false
      + id                   = (known after apply)
      + metadata             = (known after apply)
      + mount                = "secret"
      + name                 = "shop/app"
      + path                 = (known after apply)

      + custom_metadata (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  - db_user = "shop" -> null
ephemeral.vault_kv_secret_v2.db: Opening...
ephemeral.vault_kv_secret_v2.db: Opening complete after 0s
vault_kv_secret_v2.app: Creating...
vault_kv_secret_v2.app: Creation complete after 0s [id=secret/data/shop/app]
ephemeral.vault_kv_secret_v2.db: Closing...
ephemeral.vault_kv_secret_v2.db: Closing complete after 0s

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Το plan δείχνει `data_json_wo = (write-only attribute)`. Δείχνει επίσης `data = (sensitive value)`. Στο `data` ο provider αποθηκεύει ό,τι διαβάζει πίσω από το Vault. Όταν η τιμή δίνεται με write-only argument, το αφήνει άδειο. Το output `db_user` έφυγε μαζί με το `read.tf`, γι' αυτό το plan γράφει `-> null`.

TERMINAL

```
terraform state list
jq '.resources[0].instances[0].attributes | {data, data_json, data_json_wo}' \
  terraform.tfstate
```

ΕΞΟΔΟΣ

```
vault_kv_secret_v2.app
{
  "data": {},
  "data_json": null,
  "data_json_wo": null
}
```

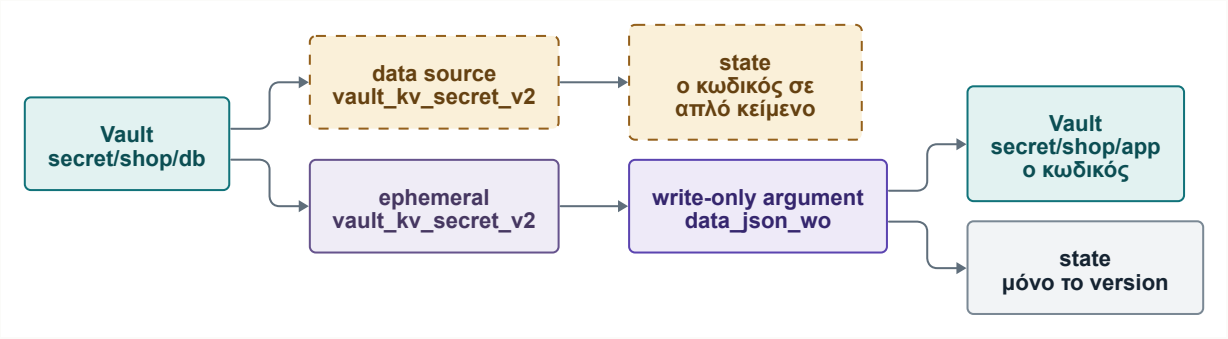
Στο state μένει μόνο το `vault_kv_secret_v2.app`. Το data source έφυγε από το state μαζί με τον κωδικό του. Τα `data` και `data_json_wo` είναι άδεια. Το secret στο Vault όμως έχει την τιμή.

TERMINAL

```
curl -s -H "X-Vault-Token: $VAULT_TOKEN" \
  $VAULT_ADDR/v1/secret/data/shop/app | jq '.data.data'
```

ΕΞΟΔΟΣ

```
{
  "DATABASE_PASSWORD": "Melomakarono-77"
}
```



Σχήμα 24.1 · Δύο δρόμοι από το Vault. Το data source αντιγράφει τον κωδικό στο state. Το ephemeral resource τον περνά στη μνήμη σε ένα write-only argument και στο state μένει μόνο το version.

Dynamic secrets

Ως τώρα ο κωδικός ήταν **στατικός**. Κάποιος τον έγραψε μία φορά και ισχύει μέχρι να τον αλλάξει κάποιος. Αν διαρρεύσει, ισχύει και για όποιον τον βρήκε.

Το Vault δίνει και **dynamic secrets**. Ένα secrets engine, για παράδειγμα το AWS secrets engine, φτιάχνει νέα credentials τη στιγμή που τα ζητάς. Κάθε σετ έρχεται με **lease**, μια άδεια με χρόνο ζωής (TTL). Όταν λήξει, το Vault ακυρώνει τα credentials στο AWS. Κάθε run παίρνει δικά του credentials και κανείς δεν κρατά μόνιμο κλειδί. Ένα κλειδί που διέρρευσε παύει να ισχύει μόλις λήξει το lease.

Ο provider `vault` έχει ephemeral resources για τέτοια credentials. Το παρακάτω απόσπασμα δεν εκτελείται στο κεφάλαιο, γιατί θέλει ρυθμισμένο AWS secrets engine. Δείχνει όμως πώς δένουν τα κομμάτια.

```hcl
ephemeral "vault_aws_access_credentials" "deploy" {
  mount = "aws"
  role  = "deploy"
}

provider "aws" {
  region     = "eu-central-1"
  access_key = ephemeral.vault_aws_access_credentials.deploy.access_key
  secret_key = ephemeral.vault_aws_access_credentials.deploy.secret_key
}
```

Το ephemeral resource ζητά νέα credentials από τον ρόλο `deploy` του engine που ζει στο mount `aws`. Ο ρόλος ορίζει στο Vault ποια δικαιώματα του AWS θα έχουν. Ο provider `aws` τα χρησιμοποιεί για όλο το run. Είναι ephemeral, οπότε δεν γράφονται στο state. Το μόνο μόνιμο μυστικό που χρειάζεται το Terraform είναι ένα token του Vault με άδεια να ζητά credentials από αυτόν τον ρόλο.

Πριν από τα ephemeral resources ο τρόπος ήταν το data source με το ίδιο όνομα, `vault_aws_access_credentials`, που υπάρχει ακόμη. Εκείνο γράφει τα credentials στο state σε απλό κείμενο, όπως κάθε data source. Λήγουν όμως μαζί με το lease, οπότε ένα state που διαρρέει αργότερα δεν δίνει πια πρόσβαση στο AWS.

Καλές πρακτικές

- Ποτέ secrets σε αρχεία `.tf` ή σε `.tfvars` που μπαίνουν σε Git. Ούτε ως `default` μεταβλητής, ούτε «προσωρινά».
- Οι τιμές φτάνουν την ώρα του run, από μεταβλητές περιβάλλοντος `TF_VAR_` ή, ακόμη καλύτερα, από secret manager όπως το Vault ή ο AWS Secrets Manager.
- Ποτέ credentials στο block `provider`. Ο AWS provider τα βρίσκει σε μεταβλητές περιβάλλοντος, profiles ή roles, όπως στο κεφάλαιο 17. Ο provider `vault` παίρνει το token από το `VAULT_TOKEN`.
- Στο HCP Terraform του κεφαλαίου 26 τα secrets μπαίνουν σε sensitive μεταβλητές του workspace, που κανείς δεν ξαναδιαβάζει από το UI ή το API. Για το cloud προτιμάς τα dynamic provider credentials, που δίνουν προσωρινά credentials σε κάθε run.
- `sensitive = true` σε κάθε μεταβλητή και output που κρατά μυστικό. Το `nonsensitive` μόνο για κομμάτια που πραγματικά δεν είναι μυστικά.
- Ephemeral resources, ephemeral μεταβλητές και write-only arguments όπου τα υποστηρίζει ο provider, ώστε το μυστικό να μη φτάσει ποτέ στο state.
- Το state και τα plan files σε backend με κρυπτογράφηση και αυστηρό έλεγχο πρόσβασης. Ποτέ σε Git.
- Ελάχιστα δικαιώματα. Το token του Vault και τα credentials του cloud που χρησιμοποιεί το Terraform έχουν μόνο όσα χρειάζεται η configuration. Όπου γίνεται, προτιμάς βραχύβια credentials όπως τα dynamic secrets.
- Ένα secret που διέρρευσε το αλλάζεις. Το σβήσιμο από το Git ή από το state δεν το παίρνει πίσω.

Σύνοψη

Μηχανισμός	Στην οθόνη	Στο state και στο plan file
`sensitive = true`	`(sensitive value)`	σε απλό κείμενο
`nonsensitive()`	η τιμή φαίνεται	σε απλό κείμενο
data source του Vault	`(sensitive value)`	σε απλό κείμενο
ephemeral μεταβλητή ή resource	δεν τυπώνεται	δεν γράφεται
write-only argument	`(write-only attribute)`	μόνο το version

Ασκήσεις

Κάθε άσκηση λύνεται σε νέο φάκελο μέσα στο `~/tfbook/ch24`, εκτός αν η εκφώνηση λέει να ξεκινήσεις από αντίγραφο άλλου. Τρέξε `terraform init` πριν από τις εντολές που ζητά η εκφώνηση. Οι ασκήσεις του Vault και του AWS θέλουν τον dev server και το Moto του κεφαλαίου.

ΑΣΚΗΣΗ 24.1 Ένα output με κωδικό

Γράψε configuration χωρίς providers με μια μεταβλητή `smtp_password` τύπου `string`, sensitive και χωρίς `default`. Ένα output `smtp_url` δίνει το string `smtp://mailer:ΚΩΔΙΚΟΣ@mail.internal:587` με τον κωδικό στη θέση του `ΚΩΔΙΚΟΣ`. Δώσε τον κωδικό `Tiropita-31` με `TF_VAR_smtp_password`. Τρέξε πρώτα plan χωρίς `sensitive` στο output και διάβασε το σφάλμα. Διόρθωσέ το, κάνε apply και δείξε τη λίστα των outputs. Τύπωσε μετά την τιμή με `-raw`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
smtp_url = <sensitive>
```

ΑΣΚΗΣΗ 24.2 Τι θυμάται το plan file

Γράψε configuration χωρίς providers με μια sensitive μεταβλητή `api_token` και ένα `terraform_data` με όνομα `deploy`, που έχει `input` την τιμή της. Δώσε το token `tok-Spanakopita-5` με `TF_VAR_api_token`. Αποθήκευσε το plan στο `deploy.tfplan` και βρες το token μέσα στο αρχείο με `terraform show -json` και `jq`, χωρίς να κάνεις apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
tok-Spanakopita-5
```

ΑΣΚΗΣΗ 24.3 Ό,τι δεν είναι μυστικό

Γράψε configuration χωρίς providers με μια sensitive μεταβλητή `db` τύπου object με δύο attributes `string`, το `user` και το `password`. Δώσε την τιμή με `TF_VAR_db` σε σύνταξη HCL, με user `shop` και password `Koulouri-9`. Γράψε ένα output `db_user` που δείχνει μόνο το user στη λίστα του `terraform output`, όχι ως `<sensitive>`. Κάνε apply.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Outputs:

db_user = "shop"
```

ΑΣΚΗΣΗ 24.4 Κωδικός για το Vault χωρίς state

Γράψε configuration με τους providers `random` 3.9.1 και `vault` 5.12.0. Ένα ephemeral `random_password` με όνομα `redis` φτιάχνει κωδικό 20 χαρακτήρων. Ένα `vault_kv_secret_v2` με όνομα `redis` τον γράφει στο secret `shop/redis` του mount `secret`, στο κλειδί `password`, με write-only argument. Κάνε apply. Δείξε με `terraform state list` τι κράτησε το state και μέτρησε με `curl` και `jq` το μήκος του κωδικού που έφτασε στο Vault.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
20
```

ΑΣΚΗΣΗ 24.5 Αλλαγή κωδικού με ephemeral μεταβλητή

Ο φάκελος της άσκησης ξεκινά ως αντίγραφο του `aws`, ώστε να έχει το lock file και να μην κατεβάσει ξανά τον AWS provider. Άλλαξε το όνομα του secret σε `shop/api-key`. Αντικατάστησε το ephemeral `random_password` με μια ephemeral μεταβλητή `api_key`, που δίνει την τιμή στο `secret_string_wo`, με version 1. Κάνε apply με `TF_VAR_api_key=key-first`. Μετά τρέξε plan με `TF_VAR_api_key=key-second` και δες τι λέει. Κάνε ό,τι χρειάζεται ώστε το Secrets Manager να κρατά το `key-second` και δείξε την τιμή του με το `aws` CLI.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
key-second
```

ΑΣΚΗΣΗ 24.6 Από το Vault στο Secrets Manager

Ο φάκελος ξεκινά πάλι ως αντίγραφο του `aws`. Γράψε με `curl` στο Vault το secret `shop/payments` με κλειδί `api_key` και τιμή `pay-Bougatsa-88`. Άλλαξε την configuration ώστε να χρησιμοποιεί τους providers `aws` και `vault`, να διαβάζει το secret με ephemeral resource και να γράφει την τιμή του `api_key` σε νέο secret του Secrets Manager με όνομα `shop/payments`, με write-only argument. Σβήσε το `password.tf`. Κάνε apply, έλεγξε ότι το state δεν έχει την τιμή και διάβασέ την από το Secrets Manager.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
pay-Bougatsa-88
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 24.1

Μια μεταβλητή `db_password` έχει `sensitive = true` και η τιμή της περνά σε ένα argument ενός resource. Ποια πρόταση είναι σωστή;

a. Το Terraform κρυπτογραφεί την τιμή πριν τη γράψει στο state.
b. Η τιμή δεν γράφεται στο state, μόνο στο plan file.
c. Το plan τη δείχνει ως `(sensitive value)`, αλλά το state την κρατά σε απλό κείμενο.
d. Η τιμή δεν μπορεί να χρησιμοποιηθεί σε συναρτήσεις όπως η `upper`.

ΕΡΩΤΗΣΗ 24.2

Ένα output `db_password` έχει `sensitive = true` και έχει γίνει apply. Τι τυπώνει η εντολή `terraform output db_password`;

a. `<sensitive>`
b. Την τιμή του output σε εισαγωγικά.
c. Σφάλμα, γιατί ένα sensitive output διαβάζεται μόνο με `-json`.
d. Τίποτα, γιατί τα sensitive outputs δεν γράφονται στο state.

ΕΡΩΤΗΣΗ 24.3

Θέλεις ένας κωδικός που δημιουργείται στο run να φτάσει στο API του cloud χωρίς να γραφτεί ποτέ στο state ή στο plan file. Ποιοι δύο μηχανισμοί χρειάζονται μαζί; Επίλεξε δύο.

a. Ένα ephemeral resource που παράγει τον κωδικό.
b. Το `sensitive = true` στο output που δείχνει τον κωδικό.
c. Ένα data source που διαβάζει τον κωδικό από το Vault.
d. Ένα write-only argument που δέχεται τον κωδικό.
e. Η συνάρτηση `nonsensitive` γύρω από τον κωδικό.

ΕΡΩΤΗΣΗ 24.4

Σωστό ή λάθος; Όταν διαβάζεις ένα secret από το Vault με data source, το Terraform δεν το αποθηκεύει στο state, γιατί το Vault το κρατά κρυπτογραφημένο.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 24.5

Ποιο είναι το βασικό πλεονέκτημα των dynamic secrets του Vault, όπως τα credentials που δίνει το AWS secrets engine;

a. Φτιάχνονται τη στιγμή που ζητούνται και λήγουν σύντομα, οπότε ένα credential που διέρρευσε παύει γρήγορα να ισχύει.
b. Αποθηκεύονται κρυπτογραφημένα στο state του Terraform.
c. Δεν χρειάζεται κανένα token για να τα ζητήσεις από το Vault.
d. Αλλάζουν αυτόματα τις τιμές σε όλα τα αρχεία `.tfvars` του repository.

ΕΡΩΤΗΣΗ 24.6

Μια ομάδα κρατά το state σε τοπικά αρχεία και τα κάνει commit σε private repository. Το state περιέχει κωδικούς βάσεων. Ποια αλλαγή είναι η σωστή;

a. `sensitive = true` σε όλες τις μεταβλητές, ώστε το state να μην έχει πια κωδικούς.
b. `terraform state rm` στα resources με κωδικούς πριν από κάθε commit.
c. Remote backend με κρυπτογράφηση και περιορισμένη πρόσβαση, το state έξω από το Git και αλλαγή των κωδικών που έμειναν στο ιστορικό.
d. Μετονομασία του `terraform.tfstate` σε `terraform.tfstate.enc` πριν από το commit.



ΚΕΦΑΛΑΙΟ 25

Troubleshooting και logging

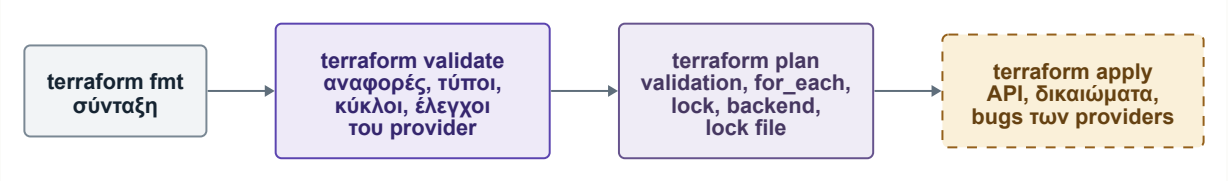
Κάθε εντολή του Terraform θα αποτύχει κάποια στιγμή. Σε αυτό το κεφάλαιο μαθαίνεις να διαβάζεις ένα σφάλμα με μέθοδο, να καταλαβαίνεις από το στάδιο όπου εμφανίστηκε τι είδους λάθος είναι και να ανοίγεις τα logs του Terraform και των providers όταν το μήνυμα δεν φτάνει. Στο τέλος ξέρεις τι κάνεις όταν καταρρέει το ίδιο το Terraform και πού αναφέρεις ένα bug.

Η μέθοδος

Ένα σφάλμα του Terraform είναι ένα **diagnostic** με σταθερή μορφή, όπως είδες στο κεφάλαιο 2. Το διαβάζεις από πάνω προς τα κάτω και κάθε μέρος απαντά σε μία ερώτηση.

1. `Error` ή `Warning`. Το πρώτο σταματά την εντολή, το δεύτερο όχι.
2. Η περίληψη, για παράδειγμα `Unsupported attribute`. Είναι σταθερή φράση και την ψάχνεις αυτούσια.
3. Η γραμμή `with`, όταν υπάρχει, λέει ποιο resource αφορά. Τη βάζουν κυρίως τα diagnostics που έρχονται από provider.
4. Η γραμμή `on` λέει αρχείο, γραμμή και block.
5. Το απόσπασμα του κώδικα με τη γραμμή που φταίει.
6. Η εξήγηση, που συχνά προτείνει και τη διόρθωση.

Μετά κάνεις δύο ερωτήσεις. Σε ποιο στάδιο εμφανίστηκε το λάθος; Ό,τι βγαίνει στο `fmt` ή στο `validate` είναι λάθος του κειμένου. Ό,τι βγαίνει μόνο στο `apply` ήρθε από τον πραγματικό κόσμο. Ποιος έγραψε το μήνυμα; Το Terraform μιλά για σύνταξη, αναφορές, state και backend. Ο provider μιλά για τα arguments των resources του και μεταφέρει τις απαντήσεις του API.



Σχήμα 25.1 · Σε ποιο στάδιο εμφανίζεται κάθε κατηγορία λάθους. Όσο πιο δεξιά εμφανίζεται ένα λάθος, τόσο πιο κοντά στον πραγματικό κόσμο βρίσκεται η αιτία του.

ΚΡΑΤΑ ΑΥΤΟ

Διάβασε ολόκληρο το πρώτο diagnostic πριν αλλάξεις κώδικα. Διόρθωσε ένα λάθος τη φορά και ξανατρέξε την ίδια εντολή, γιατί ένα λάθος συχνά κρύβει άλλα. Τα logs έρχονται τελευταία, όταν το μήνυμα δεν φτάνει.

Λάθη του κειμένου

Φτιάξε έναν φάκελο με providers που δεν χρειάζονται cloud.

TERMINAL

```
mkdir errors
cd errors
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

TERMINAL

```
terraform init
```

Γράψε δύο αρχεία με λάθη. Το `main.tf` έχει δύο λάθη σε αναφορές. Στο `notes.tf` λείπει το άγκιστρο που κλείνει το block.

main.tf

```
resource "local_file" "motd" {
  filename = "motd.txt"
  content  = "Καλημέρα\n"
}

resource "local_file" "copy" {
  filename = "copy.txt"
  content  = local_file.motd.contents
}

resource "local_file" "index" {
  filename = "index.txt"
  content  = "Δες το ${local_file.mtod.filename}\n"
}
```

notes.tf

```
resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Σημειώσεις\n"
```

Σύνταξη

Το φθηνότερο λάθος το πιάνει το `terraform fmt`, που δεν χρειάζεται καν providers.

TERMINAL

```
terraform fmt -check
echo "exit code $?"
```

ΕΞΟΔΟΣ

```
│ Error: Unclosed configuration block
│
│   on notes.tf line 1, in resource "local_file" "notes":
│    1: resource "local_file" "notes" {
│
│ There is no closing brace for this block before the end of the file. This
│ may be caused by incorrect brace nesting elsewhere in this file.

exit code 2
```

Η περίληψη `Unclosed configuration block` λέει ότι ένα block άνοιξε και δεν έκλεισε. Η γραμμή `on` δείχνει πού ξεκινά το block, όχι πού έπρεπε να κλείσει. Σε ένα μεγάλο αρχείο το άγκιστρο που λείπει μπορεί να βρίσκεται πολύ πιο κάτω. Το exit code 2 σημαίνει ότι το fmt δεν μπόρεσε να διαβάσει το αρχείο. Το 3, που είδες στο κεφάλαιο 2, σημαίνει απλώς ότι η μορφή θέλει τακτοποίηση.

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Unclosed configuration block
╵
```

Το validate βλέπει μόνο το συντακτικό λάθος. Τα δύο λάθη του `main.tf` υπάρχουν, αλλά το Terraform δεν φτάνει στον έλεγχο των αναφορών όσο δεν μπορεί να διαβάσει όλα τα αρχεία. Κλείσε το block.

notes.tf

```
resource "local_file" "notes" {
  filename = "notes.txt"
  content  = "Σημειώσεις\n"
}
```

Αναφορές και τύποι

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Unsupported attribute
│
│   on main.tf line 8, in resource "local_file" "copy":
│    8:   content  = local_file.motd.contents
│
│ This object has no argument, nested block, or exported attribute named
│ "contents". Did you mean "content"?
╵

╷
│ Error: Reference to undeclared resource
│
│   on main.tf line 13, in resource "local_file" "index":
│   13:   content  = "Δες το ${local_file.mtod.filename}\n"
│
│ A managed resource "local_file" "mtod" has not been declared in the root
│ module.
╵
```

Τώρα εμφανίστηκαν τα δύο κρυμμένα λάθη. Η σειρά των diagnostics δεν έχει σημασία και μπορεί να αλλάζει από εκτέλεση σε εκτέλεση.

- `Unsupported attribute`. Το resource `local_file.motd` υπάρχει, αλλά δεν έχει attribute `contents`. Το Terraform ξέρει από το schema του provider ποια attributes υπάρχουν και προτείνει το πιο κοντινό, `content`.
- `Reference to undeclared resource`. Το `local_file.mtod` δεν υπάρχει πουθενά στον φάκελο. Είναι ορθογραφικό λάθος στο όνομα του resource.

Και τα δύο τα πιάνει το validate, γιατί για να τα βρει αρκούν τα αρχεία και τα schemas. Διόρθωσε τις δύο αναφορές.

main.tf

```
resource "local_file" "motd" {
  filename = "motd.txt"
  content  = "Καλημέρα\n"
}

resource "local_file" "copy" {
  filename = "copy.txt"
  content  = local_file.motd.content
}

resource "local_file" "index" {
  filename = "index.txt"
  content  = "Δες το ${local_file.motd.filename}\n"
}
```

Κύκλοι

Στο ίδιο στάδιο εμφανίζονται και οι κύκλοι του κεφαλαίου 6. Πρόσθεσε δύο resources που χρειάζεται το ένα το άλλο.

loop.tf

```
resource "local_file" "ping" {
  filename = "ping.txt"
  content  = local_file.pong.filename
}

resource "local_file" "pong" {
  filename = "pong.txt"
  content  = local_file.ping.filename
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: Cycle:
│   local_file.ping
│   local_file.pong
│
╵
```

Το `Cycle` είναι από τα λίγα diagnostics χωρίς γραμμή `on`. Δεν φταίει μία γραμμή αλλά η σχέση ανάμεσα σε blocks, οπότε το Terraform απαριθμεί τα resources του κύκλου. Τον σπας αφαιρώντας μία από τις αναφορές. Εδώ σβήνεις το αρχείο.

TERMINAL

```
rm loop.tf
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

Λάθη που φαίνονται στο plan

Κάποια λάθη χρειάζονται τιμές που το validate δεν έχει. Πρόσθεσε μια μεταβλητή με validation, όπως στο κεφάλαιο 13.

variables.tf

```
variable "environment" {
  type    = string
  default = "dev"

  validation {
    condition     = contains(["dev", "prod"], var.environment)
    error_message = "To environment πρέπει να είναι dev ή prod."
  }
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

TERMINAL

```
terraform plan -var environment=test
```

ΕΞΟΔΟΣ

```
…
Terraform planned the following actions, but then encountered a problem:
…
│ Error: Invalid value for variable
│
│   on variables.tf line 1:
│    1: variable "environment" {
│      ├────────────────
│      │ var.environment is "test"
│
│ To environment πρέπει να είναι dev ή prod.
│
│ This was checked by the validation rule at variables.tf:5,3-13.
```

Το validate πέρασε, γιατί η τιμή `test` υπάρχει μόνο στη γραμμή εντολών του plan. Η γραμμή `Terraform planned the following actions, but then encountered a problem` λέει ότι το Terraform υπολόγισε κάποιες

ενέργειες αλλά μετά βρήκε πρόβλημα. Τις ενέργειες τις κρύβουμε εδώ. Ένα plan με σφάλμα δεν εφαρμόζεται ποτέ. Το diagnostic έχει δύο νέα στοιχεία. Κάτω από το απόσπασμα δείχνει την τιμή που έλεγξε, `var.environment is "test"`. Μετά το δικό σου `error_message` λέει πού βρίσκεται ο κανόνας, στο `variables.tf:5,3-13`, δηλαδή γραμμή 5, στήλες 3 έως 13. Τα preconditions και postconditions του κεφαλαίου 13 αναφέρονται με τον ίδιο τρόπο.

Το δεύτερο κλασικό λάθος του plan είναι ένα `for_each` με κλειδιά που δεν είναι γνωστά πριν από το apply, όπως στο κεφάλαιο 11.

pets.tf

```
resource "random_pet" "name" {
  count = 2
}

resource "local_file" "pet" {
  for_each = toset(random_pet.name[*].id)
  filename = "${each.key}.txt"
  content  = "${each.key}\n"
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
Terraform planned the following actions, but then encountered a problem:
…
╷
│ Error: Invalid for_each argument
│
│   on pets.tf line 6, in resource "local_file" "pet":
│    6:   for_each = toset(random_pet.name[*].id)
│     ├────────────────
│     │ random_pet.name is tuple with 2 elements
│
│ The "for_each" set includes values derived from resource attributes that
│ cannot be determined until apply, and so Terraform cannot determine the
│ full set of keys that will identify the instances of this resource.
│
│ When working with unknown values in for_each, it's better to use a map
│ value where the keys are defined statically in your configuration and where
│ only the values contain apply-time results.
│
│ Alternatively, you could use the -target planning option to first apply
│ only the resources that the for_each value depends on, and then apply a
│ second time to fully converge.
╵
```

Το `Invalid for_each argument` δεν είναι λάθος σύνταξης. Το Terraform πρέπει να ξέρει στο plan τις διευθύνσεις όλων των instances. Τα ονόματα των pets θα υπάρξουν μόνο στο apply. Η εξήγηση προτείνει και δύο λύσεις, ένα map με κλειδιά γραμμένα στη configuration ή ένα πρώτο apply με `-target`. Σβήσε το αρχείο, γιατί δεν θα το χρειαστείς άλλο.

TERMINAL

```
rm pets.tf
```

Λάθη από τον provider και το API

Τα λάθη του cloud τα βλέπεις με τον AWS provider. Φτιάξε ένα VPC με ένα subnet, όπως στο κεφάλαιο 18, αλλά με δύο λάθη.

TERMINAL

```
cd ..
mkdir network
cd network
```

versions.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}
```

main.tf

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/8"
}

resource "aws_subnet" "app" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.1.0.0/24"
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
╷
│ Error: expected "cidr_block" to contain a network Value with between 16 and 28 significant bits, got: 8
│
│   with aws_vpc.main,
│   on main.tf line 2, in resource "aws_vpc" "main":
│    2:   cidr_block = "10.0.0.0/8"
│
```

Αυτό το λάθος το έγραψε ο provider. Το καταλαβαίνεις από τη γραμμή `with aws_vpc.main,` και από τη διατύπωση, που δεν μοιάζει με τα μηνύματα του Terraform. Ο AWS provider ξέρει ότι ένα VPC δέχεται μάσκα από 16 έως 28 bits και το ελέγχει τοπικά, χωρίς να ρωτήσει το AWS. Το validate δεν καλεί ποτέ API. Διόρθωσε τη μάσκα σε `/16`.

main.tf

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "app" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.1.0.0/24"
}
```

TERMINAL

```
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_vpc.main: Creating...
aws_vpc.main: Creation complete after 0s [id=vpc-52052403c7927df24]
aws_subnet.app: Creating...
…
╷ Error: creating EC2 Subnet: operation error EC2: CreateSubnet, https response error
StatusCode: 400, RequestID: bHQsUJUhqdPxWiY2GtmyZyXsFPMg3L04FFMkKc86LQQ297GKGPt6, api
error InvalidSubnet.Range: The CIDR '10.1.0.0/24' is invalid.
…
│   with aws_subnet.app,
│   on main.tf line 5, in resource "aws_subnet" "app":
…
```

Το validate και το plan πέρασαν. Το VPC φτιάχτηκε και το subnet απέτυχε. Η περίληψη είναι μεγάλη, γιατί ο provider ενώνει όσα ξέρει.

- `creating EC2 Subnet` λέει τι προσπαθούσε να κάνει ο provider.
- `operation error EC2: CreateSubnet` λέει ποια κλήση του API απέτυχε.
- `StatusCode: 400` είναι η απάντηση HTTP. Το 400 σημαίνει ότι το αίτημα ήταν λάθος, όχι ότι το AWS είχε πρόβλημα.
- Το `RequestID` ταυτοποιεί το αίτημα. Το χρειάζεται η υποστήριξη του AWS.
- `api error InvalidSubnet.Range` είναι ο κωδικός σφάλματος του AWS και ακολουθεί το μήνυμά του. Τον κωδικό τον ψάχνεις.

Το AWS απέρριψε το subnet επειδή το `10.1.0.0/24` δεν βρίσκεται μέσα στο `10.0.0.0/16` του VPC. Αυτή τη σχέση ανάμεσα σε δύο τιμές δεν την ελέγχει ο provider. Τη βρίσκει μόνο το API, τη στιγμή της δημιουργίας.

TERMINAL

```
terraform state list
```

ΕΞΟΔΟΣ

```
aws_vpc.main
```

Το Terraform δεν γυρίζει πίσω ό,τι πέτυχε. Το VPC υπάρχει και είναι γραμμένο στο state. Μετά τη διόρθωση το apply φτιάχνει μόνο ό,τι λείπει.

main.tf

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "app" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Το πραγματικό AWS δίνει τον ίδιο κωδικό `InvalidSubnet.Range`. Ένα λάθος δικαιωμάτων έρχεται ως `UnauthorizedOperation` ή `AccessDenied`. Στο EC2 το μήνυμα έχει συχνά ένα κωδικοποιημένο κείμενο, που το διαβάζεις με `aws sts decode-authorization-message`. Τέτοια λάθη δεν τα πιάνει ούτε το validate ούτε το plan.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform validate` ελέγχει σύνταξη, αναφορές, τύπους, κύκλους και όσους ελέγχους κάνει ο provider τοπικά. Δεν πιάνει λάθη που ξέρει μόνο το API, όπως δικαιώματα, ονόματα που υπάρχουν ήδη ή τιμές που δεν ταιριάζουν μεταξύ τους. Ένα apply που αποτυγχάνει στη μέση κρατά στο state όσα πρόλαβε να φτιάξει.

Το state και το lock

Το κεφάλαιο 19 έδειξε το lock ενός remote backend. Το τοπικό state κλειδώνει με τον ίδιο τρόπο και το μήνυμα διαβάζεται ίδια. Χρειάζεσαι ένα apply που κρατά λίγο, οπότε χρησιμοποιείς το `time_sleep` του κεφαλαίου 14.

TERMINAL

```
cd ..
mkdir slow
cd slow
```

main.tf

```
terraform {
  required_providers {
    time = {
      source  = "hashicorp/time"
      version = "0.14.2"
    }
  }
}

resource "time_sleep" "wait" {
  create_duration = "20s"
}
```

TERMINAL

```
terraform init
```

Το `&` τρέχει το apply στο παρασκήνιο, σαν να το ξεκίνησε συνάδελφος από άλλο terminal. Το `until` περιμένει να ξεκινήσει η δημιουργία και μετά ζητάς plan.

TERMINAL

```
terraform apply -auto-approve > apply.log 2>&1 &
until grep -q Creating apply.log; do sleep 1; done
terraform plan
```

ΕΞΟΔΟΣ

```
Error: Error acquiring the state lock

Error message: resource temporarily unavailable
Lock Info:
  ID:        c6068fed-5b95-3611-4230-19036d318e6f
  Path:      terraform.tfstate
  Operation: OperationTypeApply
  Who:       you@laptop
  Version:   1.16.4
  Created:   2026-10-08 02:37:29.611243 +0000 UTC
  Info:


Terraform acquires a state lock to protect the state from being written
by multiple users at the same time. Please resolve the issue above and try
again. For most commands, you can disable locking with the "-lock=false"
flag, but this is not recommended.
```

Το `Error acquiring the state lock` δεν λέει ότι κάτι χάλασε. Λέει ότι κάποιος άλλος δουλεύει. Το `Error message` είναι η απάντηση του μηχανισμού που κρατά το lock. Εδώ είναι το λειτουργικό σύστημα, που λέει `resource temporarily unavailable`, ενώ στο S3 του κεφαλαίου 19 ήταν το `PreconditionFailed`. Το `Lock Info` λέει ποιος κρατά το lock. Το `Operation` είναι `OperationTypeApply`, το `Who` είναι χρήστης και μηχάνημα, το `Created` πότε πάρθηκε.

TERMINAL

```
wait
```

Το lock του τοπικού state ελευθερώνεται μόνο του όταν τελειώσει η διεργασία που το κρατά, ακόμη κι αν σκοτωθεί. Σε ένα remote backend περιμένεις ή δίνεις `-lock-timeout`. Μόνο για ορφανό lock τρέχεις `force-unlock` με το `ID`, αφού βεβαιωθείς ότι δεν τρέχει τίποτα. Το `-lock=false` δεν είναι λύση.



Logging με TF_LOG

Όταν το μήνυμα δεν φτάνει, ανοίγεις τα logs. Η μεταβλητή περιβάλλοντος `TF_LOG` ορίζει πόσο αναλυτικά γράφει το Terraform. Τα επίπεδα, από το πιο αναλυτικό στο λιγότερο, είναι τα `TRACE`, `DEBUG`, `INFO`, `WARN` και `ERROR`. Κάθε επίπεδο περιέχει και όλα όσα ακολουθούν. Το `OFF` κλείνει τα logs. Μια άγνωστη τιμή, για παράδειγμα `TF_LOG=1`, ανάβει το `TRACE` με μια προειδοποίηση.

Γύρνα στον φάκελο `errors`, που είναι πλέον σωστός. Στείλε την έξοδο του plan σε αρχείο και άφησε το log στο terminal.

TERMINAL

```
cd ../errors
TF_LOG=INFO terraform plan > plan.txt
```

ΕΞΟΔΟΣ

```
2026-10-08T05:37:50.117+0300 [INFO]  Terraform version: 1.16.4
…
2026-10-08T05:37:50.117+0300 [INFO]  CLI args: []string{"terraform", "plan"}
…
2026-10-08T05:37:50.143+0300 [INFO]  backend/local: starting Plan operation
…
2026-10-08T05:37:50.233+0300 [INFO]  backend/local: plan calling Plan
…
2026-10-08T05:37:50.264+0300 [INFO]  backend/local: plan operation completed
```

Η έξοδος του plan πήγε στο `plan.txt`. Στο terminal έμειναν μόνο οι γραμμές του log, γιατί το Terraform γράφει τα logs στο **stderr** και όχι στο stdout. Κάθε γραμμή έχει χρόνο, επίπεδο σε αγκύλες και μήνυμα. Ήδη το `INFO` λέει την έκδοση και τα ορίσματα της εντολής. Οι γραμμές `backend/local` δείχνουν ποιο backend κάνει τη δουλειά.

TF_LOG_PATH

Η `TF_LOG_PATH` γράφει τα logs σε αρχείο αντί για το stderr. Δες πόσες γραμμές δίνει κάθε επίπεδο για το ίδιο plan.

TERMINAL

```
for level in TRACE DEBUG INFO WARN ERROR; do
  TF_LOG=$level TF_LOG_PATH=$level.log terraform plan > /dev/null
  echo $level $(wc -l < $level.log)
done
```

ΕΞΟΔΟΣ

```
TRACE 1646
DEBUG 178
INFO 20
WARN 0
ERROR 0
```

Το `TRACE` δίνει πάνω από χίλιες γραμμές για ένα plan με τέσσερα αρχεία. Το `WARN` και το `ERROR` δεν έγραψαν τίποτα, γιατί το plan πέτυχε χωρίς προβλήματα. Στην πράξη ξεκινάς από το `DEBUG` ή κατευθείαν από το `TRACE` και ψάχνεις με `grep`, όπως εδώ για το lock.

```
TERMINAL

grep "statemgr.Filesystem: .*lock" TRACE.log | cut -c 30-
```

```
ΕΞΟΔΟΣ

[TRACE] statemgr.Filesystem: locking terraform.tfstate using fcntl flock
[TRACE] statemgr.Filesystem: writing lock metadata to .terraform.tfstate.lock.info
[TRACE] statemgr.Filesystem: removing lock metadata file .terraform.tfstate.lock.info
[TRACE] statemgr.Filesystem: unlocking terraform.tfstate using fcntl flock
```

Η εντολή `cut -c 30-` κόβει τον χρόνο από την αρχή της γραμμής. Το log δείχνει κάτι που δεν φαίνεται πουθενά αλλού. Το τοπικό state κλειδώνει με `fcntl flock`, ένα lock του λειτουργικού συστήματος. Δίπλα γράφεται το `.terraform.tfstate.lock.info` με τα στοιχεία που είδες στο `Lock Info`. Στο τέλος του plan το αρχείο σβήνεται και το lock ελευθερώνεται.

Δύο λεπτομέρειες της `TF_LOG_PATH` ξεγελούν συχνά. Χωρίς `TF_LOG` δεν γράφεται τίποτα, εκτός αν ανάψεις τα logs με τις `TF_LOG_CORE` και `TF_LOG_PROVIDER`, που θα δεις λίγο πιο κάτω. Και αν το αρχείο υπάρχει, τα νέα logs προστίθενται στο τέλος του.

```
TERMINAL

TF_LOG_PATH=nothing.log terraform plan > /dev/null
test -s nothing.log || echo "το nothing.log είναι άδειο"
TF_LOG=INFO TF_LOG_PATH=INFO.log terraform plan > /dev/null
grep -c "Terraform version" INFO.log
```

```
ΕΞΟΔΟΣ

το nothing.log είναι άδειο
2
```

Το `INFO.log` έχει πλέον δύο φορές τη γραμμή της έκδοσης, μία από κάθε plan. Σβήνε το αρχείο πριν από κάθε νέα δοκιμή, αλλιώς θα διαβάζεις ανακατεμένες εκτελέσεις.

TF_LOG=JSON

Η τιμή `JSON` δίνει όλα τα logs του `TRACE` με μία γραμμή JSON ανά μήνυμα. Είναι για εργαλεία που φιλτράρουν logs.

```
TERMINAL

TF_LOG=JSON TF_LOG_PATH=json.log terraform plan > /dev/null
head -1 json.log | jq .
jq -r '."@level"' json.log | sort | uniq -c
```

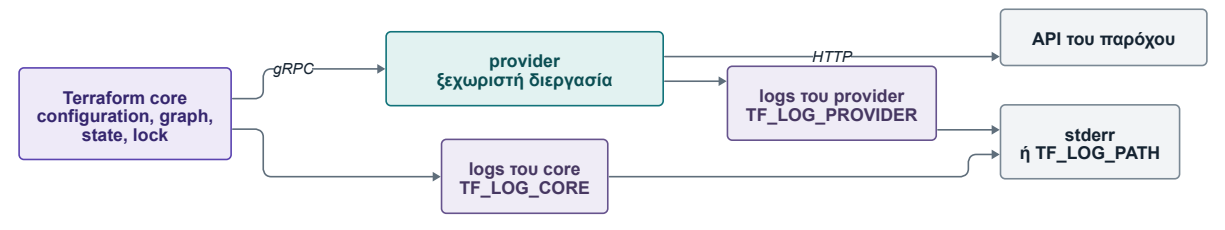
ΕΞΟΔΟΣ

```
{
  "@level": "info",
  "@message": "Terraform version: 1.16.4",
  "@timestamp": "2026-10-08T05:37:51.418919+03:00"
}
 158 debug
  20 info
1204 trace
```

Κάθε γραμμή έχει επίπεδο, μήνυμα και χρόνο σε χωριστά πεδία, οπότε το `jq` τα μετρά χωρίς `grep`. Η μορφή αυτή δεν είναι σταθερή διεπαφή και μπορεί να αλλάξει σε νεότερη έκδοση.

TF_LOG_CORE και TF_LOG_PROVIDER

Κάθε provider είναι ξεχωριστό πρόγραμμα, όπως είδες στο κεφάλαιο 5. Το Terraform τον ξεκινά ως χωριστή διεργασία και του μιλά μέσω gRPC. Τα logs έρχονται λοιπόν από δύο πηγές. Η καθεμία ανάβει με δική της μεταβλητή.



Σχήμα 25.2 · Οι δύο πηγές logs. Το Terraform core γράφει για configuration, graph και state. Κάθε provider είναι χωριστή διεργασία που γράφει για τις κλήσεις του στο API. Το TF_LOG ανάβει και τα δύο.

TERMINAL

```
TF_LOG_CORE=TRACE TF_LOG_PATH=core.log terraform plan > /dev/null
TF_LOG_PROVIDER=TRACE TF_LOG_PATH=provider.log terraform plan > /dev/null
echo core $(wc -l < core.log) provider $(wc -l < provider.log) all $(wc -l < TRACE.log)
grep -c "provider.terraform-provider-local" core.log provider.log
```

ΕΞΟΔΟΣ

```
core 1022 provider 625 all 1646
core.log:0
provider.log:519
```

Τα δύο αρχεία μαζί έχουν σχεδόν όσες γραμμές έδωσε το `TF_LOG=TRACE`. Οι γραμμές που ξεκινούν με `provider.terraform-provider-local` τις γράφει ο ίδιος ο provider και βρίσκονται μόνο στο `provider.log`. Οι μεταβλητές δέχονται τα ίδια επίπεδα με την `TF_LOG`. Όταν υποψιάζεσαι τον provider, το `TF_LOG_PROVIDER=DEBUG` σου δείχνει τι έστειλε στο API χωρίς τις εκατοντάδες γραμμές του core.

Τα logs περιέχουν μυστικά

Το κεφάλαιο 24 μέτρησε τέσσερα σημεία όπου διαρρέει ένα secret. Τα logs είναι το πέμπτο. Το `DEBUG` του AWS provider καταγράφει κάθε αίτημα HTTP μαζί με το σώμα του. Δοκίμασε το secret του κεφαλαίου 24, με ephemeral μεταβλητή και write-only argument, που δεν γράφεται ποτέ στο state.

TERMINAL

```
cd ..
mkdir secret
cd secret
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}

variable "db_password" {
  type      = string
  sensitive = true
  ephemeral = true
}

resource "aws_secretsmanager_secret" "db" {
  name = "app-db"
}

resource "aws_secretsmanager_secret_version" "db" {
  secret_id                = aws_secretsmanager_secret.db.id
  secret_string_wo         = var.db_password
  secret_string_wo_version = 1
}
```

TERMINAL

```
terraform init
```

TERMINAL

```
TF_LOG_PROVIDER=DEBUG TF_LOG_PATH=aws.log terraform apply -auto-approve -var db_password=S3cr3t-Pa55
```

ΕΞΟΔΟΣ

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

TERMINAL

```
grep -c S3cr3t-Pa55 terraform.tfstate aws.log
grep -o '"SecretString":"[^"]*"' aws.log
grep -o 'Signature=[^"]*' aws.log | sort -u
```

ΕΞΟΔΟΣ

```
terraform.tfstate:0
aws.log:1
"SecretString":"S3cr3t-Pa55"
Signature=*****
```

Το state δεν έχει τον κωδικό. Το log τον έχει καθαρό, μέσα στο σώμα του αιτήματος προς το Secrets Manager. Το `sensitive` και το `ephemeral` κρύβουν τιμές από την έξοδο και από το state, όχι από το log. Ο provider κρύβει μερικά πράγματα μόνος του, όπως την υπογραφή κάθε αιτήματος, που γράφεται `Signature=*****`. Δεν ξέρει όμως ποια δεδομένα σου είναι μυστικά.

ΠΑΓΙΔΑ

Ένα log `DEBUG` ή `TRACE` μπορεί να περιέχει κωδικούς, tokens, ονόματα λογαριασμών και ολόκληρα σώματα αιτημάτων. Μην το ανεβάζεις σε issue ή σε chat χωρίς να το διαβάσεις και να σβήσεις ό,τι είναι ευαίσθητο. Μην αφήνεις την `TF_LOG` ανοιχτή σε CI pipeline που κρατά τα logs του.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Η `TF_LOG` ανάβει τα logs με επίπεδα `TRACE`, `DEBUG`, `INFO`, `WARN` και `ERROR`, από το πιο αναλυτικό στο λιγότερο. Το `JSON` δίνει `TRACE` σε μορφή JSON. Τα logs πηγαίνουν στο stderr. Η `TF_LOG_PATH` τα γράφει σε αρχείο, αλλά δεν τα ανάβει. Τα ανάβει η `TF_LOG` ή, χωριστά για το Terraform και για τους providers, οι `TF_LOG_CORE` και `TF_LOG_PROVIDER`. Χωρίς καμία από αυτές το αρχείο μένει άδειο.

Όταν κάτι καταρρέει

Το Terraform και οι providers είναι γραμμένα στη γλώσσα Go. Ένα **panic** είναι το σφάλμα με το οποίο σταματά ένα πρόγραμμα της Go όταν βρεθεί σε κατάσταση που ο προγραμματιστής του δεν προέβλεψε. Δεν είναι λάθος της configuration σου, ακόμη κι αν το προκάλεσε κάτι μέσα της. Είναι πάντα bug.

Αν πάθει panic το ίδιο το Terraform, τυπώνει στο stderr ένα πλαίσιο που ξεκινά έτσι και ακολουθεί το stack trace, δηλαδή η αλυσίδα των συναρτήσεων που έτρεχαν τη στιγμή του panic.

ΕΞΟΔΟΣ

```
!!!!!!!!!!!!!!!!!!!!!!!!!! TERRAFORM CRASH !!!!!!!!!!!!!!!!!!!!!!!!!!!!

Terraform crashed! This is always indicative of a bug within Terraform.
Please report the crash with Terraform[1] so that we can fix this.
```

Έως και το Terraform 1.0 το panic γραφόταν μαζί με το log της εκτέλεσης σε αρχείο `crash.log` στον φάκελο εργασίας. Από την 1.1 το Terraform δεν γράφει τέτοιο αρχείο, άρα ούτε η 1.12 της εξέτασης ούτε η 1.16 του βιβλίου. Το panic και το stack trace βγαίνουν μόνο στο stderr και τα κρατάς εσύ σε αρχείο, για παράδειγμα με `2> crash.txt` στο τέλος της εντολής. Το `crash.log` θα το βρεις ακόμη σε παλιό υλικό και σε παλιές ερωτήσεις εξάσκησης. Περιγράφει την παλιά συμπεριφορά.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Στην 1.12 ένα panic του Terraform τυπώνεται στο stderr μαζί με το stack trace και δεν γράφεται αρχείο `crash.log`. Το `crash.log` ήταν το αρχείο των εκδόσεων έως την 1.0. Ένα panic είναι πάντα bug του Terraform ή του provider, ακόμη κι αν το προκάλεσε κάτι στη configuration σου.

Αν πάθει panic ένας provider, το Terraform γράφει `Stack trace from the` με το όνομα του plugin. Ακολουθεί το stack trace του provider και το σφάλμα ότι το plugin crashed. Αν η διεργασία του provider απλώς πεθάνει, για παράδειγμα επειδή τη σκότωσε το λειτουργικό σύστημα, βλέπεις κάτι άλλο. Δοκίμασέ το στον φάκελο `slow`. Το `$!` είναι το PID του apply στο παρασκήνιο και το `pkill -KILL -P` σκοτώνει τις διεργασίες που ξεκίνησε, δηλαδή τον provider.

TERMINAL

```
cd ../slow
terraform apply -auto-approve -replace=time_sleep.wait > apply.log 2>&1 &
until grep -q Creating apply.log; do sleep 1; done
pkill -KILL -P $!
wait
```

TERMINAL

```
cat apply.log
```

ΕΞΟΔΟΣ

```
time_sleep.wait: Refreshing state... [id=2026-10-08T02:37:50Z]
…
time_sleep.wait: Destroying... [id=2026-10-08T02:37:50Z]
time_sleep.wait: Destruction complete after 0s
time_sleep.wait: Creating...
╷
│ Error: Plugin did not respond
│
│   with time_sleep.wait,
│   on main.tf line 10, in resource "time_sleep" "wait":
│   10: resource "time_sleep" "wait" {
│
│ The plugin encountered an error, and failed to respond to the
│ plugin.(*GRPCProvider).ApplyResourceChange call. The plugin logs may
│ contain more details.
╵
```

Το `Plugin did not respond` λέει ότι η κλήση `ApplyResourceChange` προς τον provider δεν πήρε ποτέ απάντηση. Το μήνυμα σε στέλνει στα logs του plugin, δηλαδή στο `TF_LOG_PROVIDER`. Οι πρώτες γραμμές δείχνουν ότι το `-replace` είχε ήδη σβήσει το παλιό `time_sleep`. Το νέο δεν πρόλαβε να γραφτεί.

TERMINAL

```
terraform state list
```

Το `state list` δεν τύπωσε τίποτα. Ό,τι έγινε πριν από το crash έχει καταγραφεί. Ό,τι δεν τελείωσε δεν υπάρχει στο state.

Μετά από crash κάνεις τα εξής.

1. Κρατάς ολόκληρη την έξοδο με το stack trace.
2. Ελέγχεις με `terraform plan` τι πιστεύει τώρα το state, πριν ξανατρέξεις το apply.
3. Ψάχνεις τη φράση του panic στα issues του Terraform ή του provider. Συχνά υπάρχει ήδη διόρθωση σε νεότερη έκδοση.
4. Αν δεν υπάρχει, ανοίγεις νέο issue, όπως λέει η επόμενη ενότητα.

Έξοδος για εργαλεία

Ένα CI pipeline δεν διαβάζει πλαίσια με `|`. Το `terraform validate -json` δίνει τα ίδια diagnostics σε JSON. Ξαναγράψε ένα λάθος στον φάκελο `errors`.

TERMINAL

```
cd ../errors
echo 'output "motd" { value = local_file.motd.contents }' > outputs.tf
terraform validate -json > validate.json
echo "exit code $?"
jq '{valid, error_count, warning_count}' validate.json
jq -r '.diagnostics[] | "\(.range.filename):\(.range.start.line) \(.summary)"' validate.json
```

ΕΞΟΔΟΣ

```
exit code 1
{
  "valid": false,
  "error_count": 1,
  "warning_count": 0
}
outputs.tf:1 Unsupported attribute
```

Το exit code είναι 1, όπως και χωρίς `-json`. Το `valid` είναι `false` και κάθε diagnostic έχει `severity`, `summary`, `detail` και `range` με αρχείο, γραμμή και στήλη. Με αυτά ένα εργαλείο σημειώνει το λάθος πάνω στη γραμμή του pull request.

Το `terraform plan -json` δίνει ένα ρεύμα από γραμμές JSON, μία για κάθε γεγονός.

TERMINAL

```
rm outputs.tf
terraform plan -json | jq -r '.type + "  " + ."@message"'
```

ΕΞΟΔΟΣ

```
version  Terraform 1.16.4
planned_change  local_file.notes: Plan to create
planned_change  local_file.motd: Plan to create
planned_change  local_file.index: Plan to create
planned_change  local_file.copy: Plan to create
change_summary  Plan: 4 to add, 0 to change, 0 to destroy.
```

Το `version` λέει την έκδοση, κάθε `planned_change` είναι μία ενέργεια του plan και το `change_summary` είναι η περίληψη. Δες τι γίνεται όταν το plan αποτυγχάνει.

TERMINAL

```
terraform plan -json -var environment=test > plan.json
```

TERMINAL

```
jq -r '.type + "  " + ."@message"' plan.json
```

```
ΕΞΟΔΟΣ

version  Terraform 1.16.4
planned_change  local_file.motd: Plan to create
planned_change  local_file.notes: Plan to create
planned_change  local_file.index: Plan to create
planned_change  local_file.copy: Plan to create
change_summary  Plan: 4 to add, 0 to change, 0 to destroy.
diagnostic  Error: Invalid value for variable
```

Το ρεύμα έχει τις ίδιες ενέργειες και την ίδια περίληψη με πριν. Μόνο η τελευταία γραμμή, ένα `diagnostic` με `Error`, λέει ότι το plan απέτυχε. Ένα εργαλείο που διαβάζει μόνο την περίληψη θα νόμιζε ότι όλα πήγαν καλά. Κοιτάζεις πάντα τα diagnostics και το exit code.

Αναφορά προβλήματος

Κάθε αναφορά bug ξεκινά από το `terraform version`. Μέσα σε φάκελο που έχει περάσει από `init` λέει και τις εκδόσεις των providers.

```
TERMINAL

terraform version
```

```
ΕΞΟΔΟΣ

Terraform v1.16.4
on darwin_arm64
+ provider registry.terraform.io/hashicorp/local v2.9.1
+ provider registry.terraform.io/hashicorp/random v3.9.1
```

Η πρώτη γραμμή είναι η έκδοση του Terraform και η δεύτερη η πλατφόρμα. Οι γραμμές με `+` είναι οι providers που διάλεξε το lock file. Το `terraform version -json` δίνει τα ίδια σε JSON.

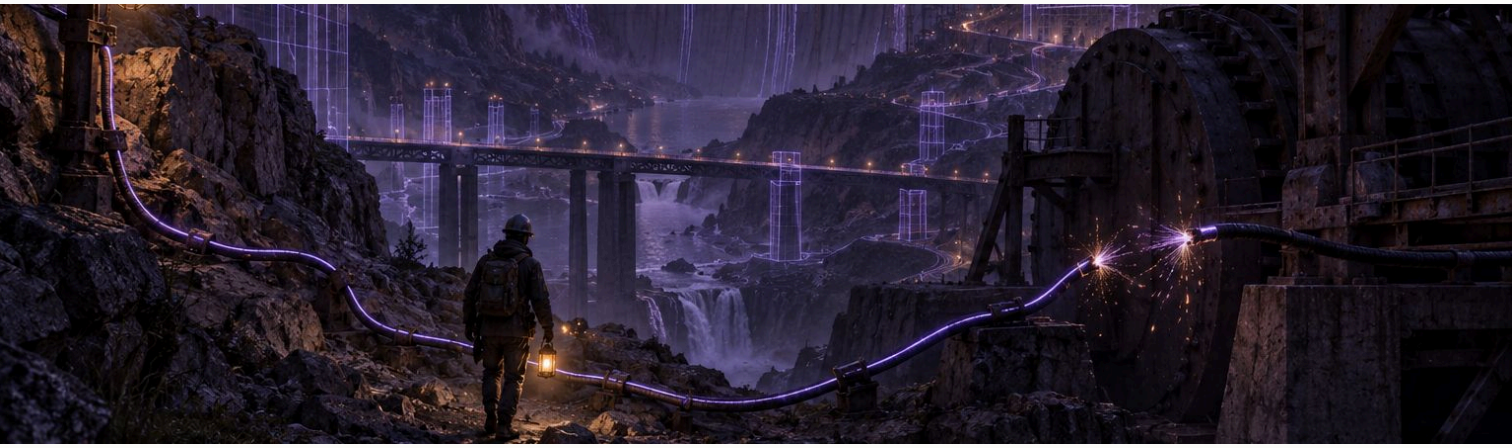
Το πού αναφέρεις το πρόβλημα το λέει συνήθως το ίδιο το μήνυμα.

- Το `TERRAFORM CRASH` και τα μηνύματα που λένε ότι πρόκειται για bug του Terraform πάνε στα issues του Terraform, στο `github.com/hashicorp/terraform`.
- Το crash ενός plugin και τα μηνύματα που γράφουν `This is a bug in the provider` πάνε στο repository του provider. Για τον AWS provider είναι το `github.com/hashicorp/terraform-provider-aws`. Η σελίδα κάθε provider στο Registry έχει link στον κώδικά του.
- Ένα σφάλμα του API, όπως το `InvalidSubnet.Range`, δεν είναι bug. Είναι διαφωνία της configuration σου με τον πραγματικό κόσμο.

Μια καλή αναφορά έχει την έξοδο του `terraform version`, τη μικρότερη configuration που δείχνει το πρόβλημα, τις εντολές που έτρεξες, ολόκληρο το diagnostic ή το crash και ένα log `TRACE`, αφού το διαβάσεις και σβήσεις ό,τι είναι ευαίσθητο.

Από το σύμπτωμα στην αιτία

Τα μηνύματα που ακολουθούν θα τα συναντήσεις ξανά και ξανά. Τα περισσότερα τα είδες σε αυτό ή σε προηγούμενα κεφάλαια.



Μήνυμα	Στάδιο	Αιτία	Τι κάνεις
`Unsupported attribute`	validate	attribute που δεν υπάρχει στο schema	διορθώνεις το όνομα
`Reference to undeclared resource`	validate	αναφορά σε resource που δεν δηλώθηκε	διορθώνεις το όνομα ή προσθέτεις το resource
`Invalid for_each argument`	plan	κλειδιά γνωστά μόνο μετά το apply	κλειδιά από τη configuration
`Inconsistent dependency lock file`	plan	το lock file δεν ταιριάζει με τους providers	`terraform init` ή `init -upgrade`
`Backend initialization required`	plan	άλλαξε το block `backend`	`terraform init` με τη σωστή σημαία
`Error acquiring the state lock`	plan, apply	άλλη εντολή κρατά το lock	περιμένεις ή `-lock-timeout`
`Saved plan is stale`	apply	το state άλλαξε μετά το `plan -out`	νέο `plan -out`
`Provider produced inconsistent result after apply`	apply	bug του provider	αναφορά στο repository του provider

Τρία από όσα δεν είδες ακόμη τα βλέπεις σε έναν νέο φάκελο με ένα resource.

TERMINAL

```
cd ..
mkdir symptoms
cd symptoms
```

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}
```

main.tf

```
resource "local_file" "motd" {
  filename = "motd.txt"
  content  = "Καλημέρα\n"
}
```

TERMINAL

```
terraform init
terraform apply -auto-approve
```

Inconsistent dependency lock file

Ένας συνάδελφος πρόσθεσε τον provider `random` και εσύ κάνεις pull χωρίς να τρέξεις `init`.

versions.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
Error: Inconsistent dependency lock file

The following dependency selections recorded in the lock file are
inconsistent with the current configuration:
  - provider registry.terraform.io/hashicorp/random: required by this configuration but no version is selected

To update the locked dependency selections to match a changed
configuration, run:
  terraform init -upgrade
```

Το lock file δεν έχει έκδοση για τον `random`. Το μήνυμα προτείνει `terraform init -upgrade`, αλλά για έναν νέο provider αρκεί το σκέτο `init`. Προσθέτει τον `random` και κρατά τον `local` όπως είναι. Το `-upgrade` θα ξαναδιάλεγε και τις εκδόσεις των υπόλοιπων providers, όπως στο κεφάλαιο 5.

TERMINAL

```
terraform init
```

ΕΞΟΔΟΣ

```
…
- Reusing previous version of hashicorp/local from the dependency lock file
- Finding hashicorp/random versions matching "3.9.1"...
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)
- Using previously-installed hashicorp/local v2.9.1
…
```

Backend initialization required

Τώρα κάποιος μετέφερε το state σε άλλο σημείο με ένα block `backend`. Το backend `local` δέχεται `path`, οπότε δεν χρειάζεσαι cloud για να το δεις.

backend.tf

```
terraform {
  backend "local" {
    path = "state/motd.tfstate"
  }
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
╷
│ Error: Backend initialization required, please run "terraform init"
│
│ Reason: Initial configuration of the requested backend "local"
│
│ The "backend" is the interface that Terraform uses to store state,
…
```

Το Terraform θυμάται στον φάκελο `.terraform` ποιο backend ρύθμισε το τελευταίο `init`. Αν η configuration λέει άλλο, σταματά πριν αγγίξει το state. Το `init -migrate-state` του κεφαλαίου 19 ρυθμίζει το νέο backend και ρωτά αν θα αντιγράψει το υπάρχον state.

TERMINAL

```
terraform init -migrate-state
```

ΕΞΟΔΟΣ

```
…
Do you want to copy existing state to the new backend?
…
  configured "local" backend. Do you want to copy this state to the new "local"
  backend? Enter "yes" to copy and "no" to start with an empty state.
…
  Enter a value: yes
…
Terraform has been successfully initialized!
…
```

TERMINAL

```
ls state
```

ΕΞΟΔΟΣ

```
motd.tfstate
```

Απάντησες `yes` και το state αντιγράφηκε στο `state/motd.tfstate`, που ορίζει το νέο backend.

Saved plan is stale

Ένα αποθηκευμένο plan, όπως στο κεφάλαιο 3, δεν ισχύει αν το state άλλαξε μετά το `plan -out`.

TERMINAL

```
sed -i.bak 's/Καλημέρα/Καλησπέρα/' main.tf
terraform plan -out=motd.tfplan > /dev/null
terraform apply -auto-approve > /dev/null
terraform apply motd.tfplan
```

ΕΞΟΔΟΣ

```
Error: Saved plan is stale

The given plan file can no longer be applied because the state was changed
by another operation after the plan was created.
```

Το `apply -auto-approve` άλλαξε το state, οπότε το `motd.tfplan` περιγράφει αλλαγές πάνω σε state που δεν υπάρχει πια. Φτιάχνεις νέο plan.

Provider produced inconsistent result after apply

Αυτό το μήνυμα δεν μπορείς να το προκαλέσεις με σωστό provider. Μετά από ένα apply ο provider δηλώνει ότι το resource έχει μια τιμή διαφορετική από αυτή που υποσχέθηκε στο plan. Το μήνυμα λέει ότι ο provider `produced an unexpected new value` και κλείνει λέγοντας ότι πρόκειται για bug του provider, που αναφέρεται στα issues του provider. Το resource συνήθως φτιάχτηκε. Τρέχεις `terraform plan` για να δεις τι πιστεύει το state, ψάχνεις στα issues του provider και δοκιμάζεις νεότερη έκδοση του provider.

Ασκήσεις

Λύσε κάθε άσκηση σε νέο φάκελο μέσα στο `~/tfbook/ch25`. Τρέξε `terraform init` πριν από τις άλλες εντολές.

ΑΣΚΗΣΗ 25.1 Log σε αρχείο

Γράψε configuration με ένα `local_file` που φτιάχνει το `hello.txt`. Τρέξε `terraform plan` έτσι ώστε το terminal να δείχνει μόνο το plan και το log επιπέδου `INFO` να γραφτεί στο `plan.log`. Μετά τύπωσε από το log μόνο το επίπεδο και την έκδοση του Terraform, χωρίς τον χρόνο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
[INFO]  Terraform version: 1.16.4
```

ΑΣΚΗΣΗ 25.2 Core ή provider

Στην configuration της προηγούμενης άσκησης γράψε δύο logs επιπέδου `TRACE` για το ίδιο plan. Το `core.log` να έχει μόνο τα logs του Terraform και το `provider.log` μόνο του provider. Με `grep -l` δείξε ποιο αρχείο έχει γραμμές `backend/local` και ποιο γραμμές `provider.terraform-provider-local`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
core.log
provider.log
```

ΑΣΚΗΣΗ 25.3 Diagnostics για CI

Γράψε το παρακάτω `main.tf` και τρέξε `terraform init`. Μετά γράψε μία εντολή που τυπώνει τον αριθμό των σφαλμάτων και μία που τυπώνει κάθε diagnostic ως `αρχείο:γραμμή περίληψη`, ταξινομημένα κατά γραμμή.

main.tf

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "2.9.1"
    }
  }
}

resource "local_file" "a" {
  filename = "a.txt"
  content  = "a\n"
}

resource "local_file" "b" {
  filename = "b.txt"
  content  = local_file.a.contents
}

resource "local_file" "c" {
  filename = "c.txt"
  content  = local_file.d.content
}
```

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
2
main.tf:17 Unsupported attribute
main.tf:22 Reference to undeclared resource
```

ΑΣΚΗΣΗ 25.4 Τι θα κάνει το plan

Γράψε configuration με δύο `local_file`, το `a` και το `b`. Με `terraform plan -json` και `jq` τύπωσε για κάθε προγραμματισμένη αλλαγή την ενέργεια και τη διεύθυνση, ταξινομημένες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
create local_file.a
create local_file.b
```

ΑΣΚΗΣΗ 25.5 Νέος provider χωρίς init

Γράψε configuration με τον provider `local` και ένα `local_file`, τρέξε `init` και κάνε apply. Μετά πρόσθεσε τον provider `random` στο `required_providers` και ένα `random_pet`. Τρέξε `terraform plan`, διόρθωσε το λάθος χωρίς να αλλάξεις έκδοση στον `local` και δείξε με `terraform version` τους providers του φακέλου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
+ provider registry.terraform.io/hashicorp/local v2.9.1
+ provider registry.terraform.io/hashicorp/random v3.9.1
```

ΑΣΚΗΣΗ 25.6 Ποια εντολή το πιάνει

Γράψε configuration με τον AWS provider, ένα `aws_vpc` με `cidr_block` `10.0.0.0/8` και ένα `aws_subnet` μέσα του με `192.168.1.0/24`. Τρέξε `terraform validate`. Διόρθωσε μόνο ό,τι αναφέρει και κάνε apply. Διόρθωσε ό,τι αναφέρει το apply και ξανακάνε apply. Πόσα resources φτιάχνει το τελευταίο apply;

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

ΑΣΚΗΣΗ 25.7 Validation σε JSON

Γράψε configuration με μια μεταβλητή `port` τύπου `number` και ένα `local_file` που γράφει τη θύρα στο `app.conf`. Η μεταβλητή να δέχεται μόνο τιμές από 1024 έως 65535, με δικό σου μήνυμα λάθους. Τρέξε `terraform plan` με `-json` και `-var port=80` και τύπωσε με `jq` το `severity`, το `summary` και το `detail` κάθε diagnostic.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
error
Invalid value for variable
Η θύρα πρέπει να είναι από 1024 έως 65535.

This was checked by the validation rule at main.tf:13,3–13.
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 25.1

Ένα apply αποτυγχάνει με μήνυμα που δεν εξηγεί την αιτία. Ποια τιμή της `TF_LOG` δίνει τα πιο αναλυτικά logs;

a. `DEBUG`
b. `INFO`
c. `TRACE`
d. `VERBOSE`

ΕΡΩΤΗΣΗ 25.2

Σωστό ή λάθος; Αν ορίσεις μόνο την `TF_LOG_PATH=terraform.log`, το Terraform γράφει αναλυτικά logs σε αυτό το αρχείο.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 25.3

Υποψιάζεσαι ότι ο AWS provider στέλνει λάθος τιμή στο API. Θέλεις τα logs του provider χωρίς τις εκατοντάδες γραμμές του Terraform core. Τι ορίζεις;

a. `TF_LOG_PROVIDER=DEBUG`
b. `TF_LOG=PROVIDER`
c. `TF_LOG_PATH=provider.log`
d. `TF_LOG_CORE=DEBUG`

ΕΡΩΤΗΣΗ 25.4

Ποια δύο προβλήματα δεν εντοπίζει το `terraform validate`; Επίλεξε δύο.

a. Αναφορά σε resource που δεν έχει δηλωθεί.
b. Subnet με CIDR έξω από το εύρος του VPC του.
c. Δύο resources που αναφέρονται το ένα στο άλλο.
d. Χρήστης χωρίς δικαίωμα να δημιουργήσει instances.
e. Block που ανοίγει και δεν κλείνει.

ΕΡΩΤΗΣΗ 25.5

Ένα apply τελειώνει με `Provider produced inconsistent result after apply` και το μήνυμα γράφει `This is a bug in the provider`. Ποια είναι η σωστή κίνηση;

a. Σβήνεις το `terraform.tfstate` και ξανακάνεις apply.
b. Ορίζεις `-lock=false` και ξανακάνεις apply.
c. Ελέγχεις με plan το state και αναφέρεις το bug στο repository του provider.
d. Ανοίγεις issue στο repository του Terraform.

ΕΡΩΤΗΣΗ 25.6

Ένας συνάδελφος θέλει να επισυνάψει σε δημόσιο issue το log ενός apply με `TF_LOG=DEBUG`. Όλες οι μεταβλητές με κωδικούς είναι `sensitive`. Τι πρέπει να κάνει πρώτα;

a. Να διαβάσει το log και να σβήσει κωδικούς και άλλα ευαίσθητα δεδομένα.
b. Τίποτα, γιατί οι τιμές `sensitive` δεν γράφονται ποτέ στα logs.
c. Να το ξαναβγάλει με `TF_LOG=JSON`, που κρύβει τα μυστικά.
d. Να το ξαναβγάλει με `TF_LOG_PATH`, που κρυπτογραφεί το αρχείο.



ΜΕΡΟΣ VI

Ομάδα και εξέταση

Ως τώρα δούλευες μόνος σου. Το state ζούσε σε αρχείο ή σε bucket και κάθε apply έτρεχε στο δικό σου μηχάνημα. Σε αυτό το μέρος βλέπεις πώς δουλεύει μια ομάδα με το HCP Terraform, χτίζεις ένα μικρό project από την αρχή με όσα έμαθες και κλείνεις με μια προσομοίωση της εξέτασης.



ΚΕΦΑΛΑΙΟ 26

HCP Terraform

Το HCP Terraform τρέχει το Terraform για λογαριασμό μιας ομάδας. Κρατά state, μεταβλητές και ιστορικό, εκτελεί plan και apply σε δικά του μηχανήματα και ελέγχει κάθε αλλαγή με πολιτικές πριν από το apply. Σε αυτό το κεφάλαιο συνδέεις τη γραμμή εντολών σου με το block `cloud` και γνωρίζεις τα εργαλεία συνεργασίας και ελέγχου που ζητά η εξέταση.

> **ΣΗΜΕΙΩΣΗ**
> Το κεφάλαιο θέλει λογαριασμό, οπότε δεν εκτελέστηκε κατά το build. Ο κώδικας HCL ελέγχθηκε με `terraform fmt`. Η έξοδος που δείχνουμε είναι ενδεικτική, όπως γράφει η ετικέτα της. Για να δοκιμάσεις, φτιάξε δωρεάν λογαριασμό στο `app.terraform.io` και βάλε το δικό σου organization στη θέση του `acme`.

Από το laptop στην ομάδα

Στο κεφάλαιο 19 έβαλες το state σε S3 με lock. Μένουν όμως άλλα προβλήματα. Ο καθένας κάνει apply από το laptop του, με δικά του credentials και δική του έκδοση του Terraform. Τα `.tfvars` κυκλοφορούν με email. Κανείς δεν ξέρει ποιος έκανε το τελευταίο apply ούτε τι έδειχνε το plan του.

Το **HCP Terraform** είναι υπηρεσία της HashiCorp που λύνει αυτά τα προβλήματα. Το HCP σημαίνει HashiCorp Cloud Platform. Παλιότερα λεγόταν Terraform Cloud, όνομα που θα βρεις σε παλιά κείμενα και στο πρόθεμα `TFC_` κάποιων μεταβλητών.

Έκδοση	Τι είναι	Πού τρέχει
Terraform Community Edition	Το δωρεάν CLI του βιβλίου	Στο μηχάνημά σου ή στο CI σου

Έκδοση	Τι είναι	Πού τρέχει
HCP Terraform	Υπηρεσία SaaS στο `app.terraform.io`	Στις υποδομές της HashiCorp
Terraform Enterprise	Η ίδια πλατφόρμα σε δική σου εγκατάσταση	Στο δίκτυό σου, ακόμη και χωρίς internet

Το **Terraform Enterprise** είναι η self-hosted μορφή του HCP Terraform. Το διαλέγουν οργανισμοί με αυστηρές απαιτήσεις ασφάλειας και συμμόρφωσης. Οι έννοιες είναι κοινές. Για τον κώδικά σου αλλάζει μόνο το hostname.

Το HCP Terraform έχει τα πλάνα Free, Essentials, Standard και Premium, ανά organization. Το Free διαχειρίζεται έως 500 managed resources χωρίς χρέωση. Αρκετές λειτουργίες αυτού του κεφαλαίου θέλουν πληρωμένο πλάνο.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Οι στόχοι της εξέτασης δεν περιλαμβάνουν τιμές και πλάνα, αλλά τι κάνει κάθε λειτουργία. Αν μια ερώτηση γράφει Terraform Cloud, εννοεί το HCP Terraform.

Organizations, projects και workspaces

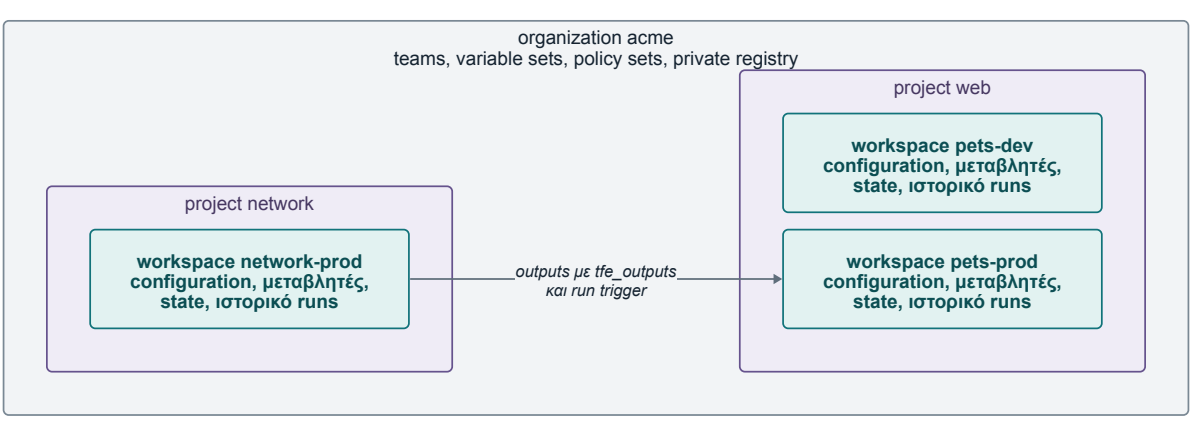
Το **organization** είναι ο κοινός χώρος μιας εταιρείας. Εκεί ζουν οι χρήστες και οι teams, οι κοινές μεταβλητές, οι πολιτικές και το private registry. Κάθε organization έχει την team `owners` με πλήρη δικαιώματα.

Το **project** είναι ένας φάκελος από workspaces. Κάθε workspace ανήκει σε ακριβώς ένα project. Αν δεν διαλέξεις, μπαίνει στο `Default Project`. Μια team με πρόσβαση σε ένα project αποκτά πρόσβαση σε όλα τα workspaces του.

Το **workspace** κρατά ό,τι χρειάζεται το Terraform για μία συλλογή υποδομής. Λειτουργεί όπως ένας χωριστός φάκελος εργασίας.

Συστατικό	Terraform στον φάκελό σου	Workspace στο HCP Terraform
Configuration	Αρχεία στον δίσκο	Από Git ή ανέβασμα από CLI και API
Τιμές μεταβλητών	`.tfvars`, `-var`, περιβάλλον	Αποθηκευμένες στο workspace
State	Αρχείο ή backend όπως το S3	Στο workspace, με όλες τις παλιές εκδόσεις
Credentials	Το περιβάλλον του shell σου	Sensitive μεταβλητές ή dynamic credentials

Το workspace κρατά επίσης το ιστορικό κάθε run. Η HashiCorp προτείνει μικρά workspaces με μοναδικά ονόματα, όπως `network-prod` και `app-prod`.



Σχήμα 26.1 · Η ιεραρχία στο HCP Terraform. Το organization κρατά ό,τι μοιράζονται όλοι. Τα projects ομαδοποιούν workspaces για δικαιώματα. Κάθε workspace έχει τη δική του configuration, μεταβλητές, state και ιστορικό και μπορεί να διαβάσει outputs από άλλο workspace.

ΠΑΓΙΔΑ

Τα **CLI workspaces** του κεφαλαίου 20 είναι πολλά states της ίδιας configuration, στο ίδιο backend και με τα ίδια credentials. Είναι προαιρετικά. Τα **workspaces του HCP Terraform** είναι υποχρεωτικά και το καθένα έχει δική του configuration, μεταβλητές, δικαιώματα και ιστορικό. Μοιάζουν με χωριστούς φακέλους, όχι με CLI workspaces.

Runs και workflows

Ένα **run** είναι ένα plan και, αν εγκριθεί, το apply του, πάντα μέσα σε ένα workspace. Κάθε workspace έχει δική του ουρά. Ένα run τη φορά προχωρά και όσο τρέχει κλειδώνει το workspace. Η configuration και οι μεταβλητές ενός run παγώνουν τη στιγμή που ξεκινά. Τα runs ξεκινούν με τρία workflows.

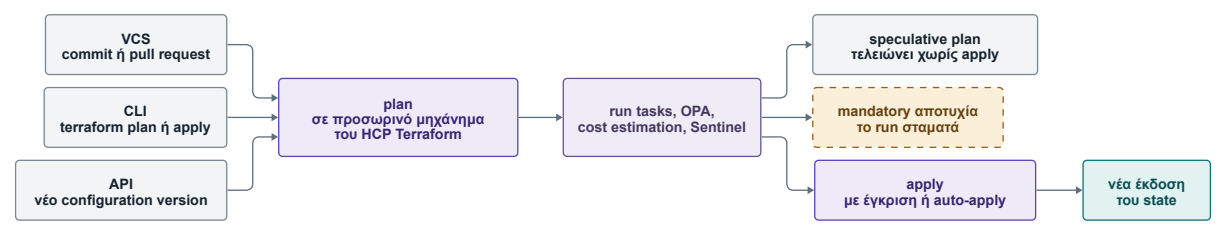
1. **VCS-driven.** Το workspace συνδέεται με ένα κλαδί ενός repository, για παράδειγμα στο GitHub. Το HCP Terraform γράφει webhooks στο repository. Κάθε merge στο κλαδί ξεκινά run. Κάθε pull request προς το κλαδί ξεκινά **speculative plan** και το αποτέλεσμα φαίνεται μέσα στο pull request. Είναι ο κύριος τρόπος εργασίας.



2. **CLI-driven.** Τρέχεις `terraform plan` ή `terraform apply` στον φάκελό σου. Το Terraform ανεβάζει τα αρχεία και το run τρέχει απομακρυσμένα, με την έξοδο ζωντανά στο terminal σου.
3. **API-driven.** Ένα δικό σου εργαλείο, συνήθως pipeline CI, πακετάρει την configuration σε `.tar.gz` και την ανεβάζει ως νέο **configuration version** μέσω API. Το run ξεκινά αμέσως. Ταιριάζει σε σύστημα εκδόσεων που δεν υποστηρίζεται ή σε configuration που παράγεται από πρόγραμμα.

Ένα speculative plan είναι run μόνο με plan. Δείχνει τι θα άλλαζε και ποιες πολιτικές επηρεάζονται, αλλά δεν γίνεται ποτέ apply και δεν περιμένει στην ουρά. Ξεκινά από pull request, από `terraform plan` στο CLI ή από το API.

Ένα κανονικό plan περιμένει έγκριση από χρήστη με δικαίωμα apply, εκτός αν το workspace έχει **auto-apply**.



Σχήμα 26.2 · Η διαδρομή ενός run. Και οι τρεις πηγές καταλήγουν στο ίδιο plan. Ακολουθούν οι έλεγχοι και μόνο τότε το apply γράφει νέα έκδοση του state. Ένα speculative plan περνά από τους ελέγχους και σταματά πριν από το apply.

Execution modes

Το **execution mode** λέει πού τρέχει το Terraform. Ορίζεται ανά workspace, με προεπιλογή αυτή του project και του organization.

- **Remote.** Η προεπιλογή. Κάθε run τρέχει σε προσωρινό μηχάνημα του HCP Terraform.
- **Local.** Το plan και το apply τρέχουν στο μηχάνημά σου. Το workspace κρατά μόνο το state. Οι μεταβλητές του workspace και τα variable sets **δεν χρησιμοποιούνται**. Λειτουργίες όπως οι πολιτικές, το cost estimation και τα notifications δεν υπάρχουν σε αυτό το mode.
- **Agent.** Το run τρέχει σε έναν **HCP Terraform agent**, ένα πρόγραμμα που εγκαθιστάς στο δίκτυό σου. Ο agent ρωτά το HCP Terraform για δουλειά με εξερχόμενη σύνδεση. Έτσι διαχειρίζεσαι ιδιωτικά δίκτυα και υποδομή on premises, όπως vSphere, χωρίς εισερχόμενη πρόσβαση από το internet.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Σε local execution mode το HCP Terraform κρατά μόνο το state. Για runs μέσα σε ιδιωτικό δίκτυο χωρίς εισερχόμενες συνδέσεις η απάντηση είναι οι agents.

Σύνδεση από τη γραμμή εντολών

Για το CLI-driven workflow χρειάζεσαι ένα token, ένα block `cloud` και ένα `terraform init`.

terraform login

Το `terraform login` παίρνει ένα API token και το αποθηκεύει. Χωρίς όρισμα συνδέεται στο `app.terraform.io`. Για Terraform Enterprise γράφεις το hostname, για παράδειγμα `terraform login tfe.acme.com`.

TERMINAL

```
terraform login
```

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /home/you/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value: yes


---------------------------------------------------------------------------

Terraform must now open a web browser to the tokens page for app.terraform.io.

If a browser does not open this automatically, open the following URL to proceed:
    https://app.terraform.io/app/settings/tokens?source=terraform-login


---------------------------------------------------------------------------

Generate a token using your browser, and copy-paste it into this prompt.

Terraform will store the token in plain text in the following file
for use by subsequent commands:
    /home/you/.terraform.d/credentials.tfrc.json

Token for app.terraform.io:
  Enter a value:
```

Το Terraform λέει πού θα γράψει το token και ζητά `yes`. Μετά ανοίγει τη σελίδα των tokens, όπου φτιάχνεις ένα token και το επικολλάς. Το αρχείο είναι απλό JSON.

~/.terraform.d/credentials.tfrc.json

```
{
  "credentials": {
    "app.terraform.io": {
      "token": "xxxxxx.atlasv1.zzzzzzzzzzzzz"
    }
  }
}
```

Σε ένα pipeline δεν υπάρχει browser. Εκεί δίνεις το token με τη μεταβλητή `TF_TOKEN_` και το hostname, με κάτω παύλες στη θέση των τελειών. Αν υπάρχει και το αρχείο, κερδίζει η μεταβλητή.

TERMINAL

```
export TF_TOKEN_app_terraform_io="$HCP_TOKEN"
```

Για το CLI χρειάζεσαι user token ή team token, όχι organization token. Το `terraform logout` σβήνει το token από τον δίσκο, αλλά δεν το ακυρώνει.

> **ΠΑΓΙΔΑ**
> Το `credentials.tfrc.json` κρατά το token σε απλό κείμενο. Μην το βάλεις σε Git. Το block `cloud` δέχεται και argument `token`, αλλά τότε το token καταλήγει στο repository. Προτίμησε το `terraform login` ή το `TF_TOKEN_`.

Το block cloud

Το block `cloud` μπαίνει μέσα στο block `terraform` και λέει σε ποιο organization και σε ποιο workspace ανήκει ο φάκελος. Το παράδειγμα χρησιμοποιεί τον provider `random`, που δεν θέλει credentials.

TERMINAL

```
mkdir pets
cd pets
```

main.tf

```
terraform {
  cloud {
    organization = "acme"

    workspaces {
      name = "pets-dev"
    }
  }

  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}

resource "random_pet" "server" {
  length = 2
}

output "server_name" {
  value = random_pet.server.id
}
```

Το `organization` είναι το όνομα του organization και το block `workspaces` διαλέγει το workspace `pets-dev`. Το argument `hostname` έχει προεπιλογή το `app.terraform.io` και το γράφεις μόνο για Terraform Enterprise. Ισχύουν οι κανόνες του backend από το κεφάλαιο 19 και κάποιοι ακόμη.

- Μία configuration έχει το πολύ ένα block `cloud`.
- Το `cloud` και το `backend` δεν συνυπάρχουν.
- Οι τιμές είναι σταθερές. Μεταβλητές και locals δίνουν το σφάλμα `Variables not allowed`.
- Κάθε αλλαγή του block θέλει νέο `terraform init`.

name ή tags

Το block `workspaces` διαλέγει workspaces με δύο τρόπους. Γράφεις τον έναν ή τον άλλον, όχι και τους δύο.

Με **`name`** ο φάκελος δουλεύει με ένα μόνο workspace. Αν δεν υπάρχει, το `terraform init` το δημιουργεί. Οι εντολές `terraform workspace` δεν διαχειρίζονται workspaces σε αυτή τη μορφή.

Με **`tags`** ο φάκελος συνδέεται με όλα τα workspaces που έχουν αυτά τα tags. Τα `terraform workspace list` και `select` δουλεύουν με αυτά και το `terraform workspace new` φτιάχνει νέο workspace με τα ίδια tags. Έτσι μία configuration εξυπηρετεί τα `pets-dev` και `pets-prod`.

HCL

```
terraform {
  cloud {
    organization = "acme"

    workspaces {
      project = "web"
      tags = {
        app = "pets"
      }
    }
  }
}
```

Η μορφή map με κλειδί και τιμή υπάρχει από το Terraform 1.10. Η παλιά μορφή είναι λίστα, για παράδειγμα `tags = ["pets"]`. Το προαιρετικό `project` περιορίζει την αναζήτηση σε ένα project, όπου μπαίνουν και τα νέα workspaces. Αν γράψεις και `name` και `tags`, το init σταματά.

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Initializing HCP Terraform...

Error: Invalid workspaces configuration

  on main.tf line 2, in terraform:
   2:   cloud {

Only one of workspace "tags" or "name" is allowed.
...
```

Αν δεν γράψεις κανένα από τα δύο και δεν ορίσεις `TF_WORKSPACE`, το μήνυμα είναι `Missing workspace mapping strategy`.

Οι τιμές του block μπορούν να έρθουν και από το περιβάλλον. Η τιμή του block κερδίζει τη μεταβλητή. Το block `cloud` πρέπει να υπάρχει, έστω και άδειο.

Μεταβλητή	Αντικαθιστά
`TF_CLOUD_ORGANIZATION`	`organization`
`TF_CLOUD_HOSTNAME`	`hostname`
`TF_CLOUD_PROJECT`	`workspaces.project`
`TF_WORKSPACE`	`workspaces.name`, μόνο για workspace που ήδη υπάρχει

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το block `cloud` υπάρχει από το Terraform 1.1. Πριν από αυτό η σύνδεση γινόταν με `backend "remote"`. Για να περάσεις στο `cloud`, αλλάζεις το block και τρέχεις `terraform init`. Μάθε και τους άκυρους συνδυασμούς. Το `name` με τα `tags`, το `cloud` με `backend` και μεταβλητές μέσα στο block.

terraform init και μεταφορά του state

Μετά το login τρέχεις το init στον φάκελο `pets`.

TERMINAL

```
terraform init
```

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Initializing HCP Terraform...
Initializing provider plugins...
- Finding hashicorp/random versions matching "3.9.1"...
- Installing hashicorp/random v3.9.1...
- Installed hashicorp/random v3.9.1 (signed by HashiCorp)
...
HCP Terraform has been successfully initialized!

You may now begin working with HCP Terraform. Try running "terraform plan" to
see any changes that are required for your infrastructure.

If you ever set or change modules or Terraform Settings, run "terraform init"
again to reinitialize your working directory.
```

Η πρώτη ενότητα λέει `Initializing HCP Terraform` αντί για `Initializing the backend`. Αν το `pets-dev` δεν υπήρχε, το init το έφτιαξε. Ένα workspace που φτιάχνει το CLI παίρνει το execution mode της προεπιλογής, συνήθως το remote. Παίρνει επίσης την ίδια έκδοση του Terraform με το CLI σου, αν το HCP Terraform την προσφέρει. Δεν έχει δικές του μεταβλητές. Ισχύουν μόνο τα variable sets του organization και του project του.

Αν ο φάκελος έχει ήδη state, τοπικό ή στο S3, το init προτείνει να το μεταφέρει.

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Initializing HCP Terraform...
Do you wish to proceed?
  As part of migrating to HCP Terraform, Terraform can optionally copy
  your current workspace state to the configured HCP Terraform workspace.

  Answer "yes" to copy the latest state snapshot to the configured
  HCP Terraform workspace.

  Answer "no" to ignore the existing state and just activate the configured
  HCP Terraform workspace with its existing state, if any.

  Should Terraform migrate your existing state?

  Enter a value: yes
```

Με `yes` το state γίνεται η πρώτη έκδοση στην καρτέλα States του workspace. Μετά σβήνεις το τοπικό `terraform.tfstate`, για να μην το χρησιμοποιήσει κανείς. Ένα state στο S3 μένει στο bucket, αλλά δεν διαβάζεται πια.

ΠΑΓΙΔΑ

Στο κεφάλαιο 19 άλλαζες backend με `terraform init -migrate-state`. Προς το HCP Terraform αυτή η σημαία απορρίπτεται, όπως και η `-reconfigure`. Τρέχεις σκέτο `terraform init`.

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Error: Invalid command-line option

The -migrate-state option is for migration between state backends only, and
is not applicable when using HCP Terraform.

HCP Terraform migrations have additional steps, configured by interactive
prompts.
```

Ένα run από το CLI

Με το workspace έτοιμο τρέχεις plan όπως πάντα.

TERMINAL

```
terraform plan
```

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Running plan in HCP Terraform. Output will stream here. Pressing Ctrl-C
will stop streaming the logs, but will not stop the plan running remotely.

Preparing the remote plan...

To view this run in a browser, visit:
https://app.terraform.io/app/acme/pets-dev/runs/run-9qKfWm2xTzR4bVdE

Waiting for the plan to start...

Terraform v1.16.4
on linux_amd64
Initializing plugins and modules...

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # random_pet.server will be created
  + resource "random_pet" "server" {
      + id        = (known after apply)
      + length    = 2
      + separator = "-"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + server_name = (known after apply)
```

Το Terraform ανέβασε τον φάκελο, ζήτησε run και μεταφέρει την έξοδο. Το `linux_amd64` είναι η πλατφόρμα του προσωρινού μηχανήματος, όχι του laptop σου. Η διεύθυνση ανοίγει τη σελίδα του run για όλη την ομάδα. Το `Ctrl-C` σταματά μόνο τη μεταφορά της εξόδου. Αυτό το plan είναι speculative. Το `terraform apply` ξεκινά κανονικό run.

TERMINAL

```
terraform apply
```

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Running apply in HCP Terraform. Output will stream here. Pressing Ctrl-C
will cancel the remote apply if it's still pending. If the apply started it
will stop streaming the logs, but will not stop the apply running remotely.

Preparing the remote apply...
...
Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + server_name = (known after apply)

Do you want to perform these actions in workspace "pets-dev"?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

random_pet.server: Creating...
random_pet.server: Creation complete after 0s [id=quick-falcon]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

server_name = "quick-falcon"
```

Η ερώτηση λέει σε ποιο workspace εφαρμόζεται η αλλαγή. Το state έγινε νέα έκδοση στο workspace, όχι αρχείο στον φάκελό σου.

Το CLI ανεβάζει όλο τον φάκελο εκτός από τα `.git` και `.terraform`. Ό,τι άλλο θέλεις να εξαιρέσεις το γράφεις στο `.terraformignore`. Σε workspace με VCS το `terraform plan` από το CLI επιτρέπεται, το `terraform apply` όχι. Εκεί οι αλλαγές περνούν με merge.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Με το block `cloud` το CLI γίνεται τηλεχειριστήριο. Γράφεις τις ίδιες εντολές, αλλά το Terraform εκτελείται στο workspace, με τις μεταβλητές, τα credentials και το state του workspace.

Μεταβλητές και variable sets

Οι μεταβλητές ενός workspace έχουν δύο κατηγορίες.

- **Terraform variables.** Δίνουν τιμές στις input variables, όπως ένα `.tfvars`. Με το πεδίο HCL η τιμή γράφεται σε HCL, για παράδειγμα λίστα.
- **Environment variables.** Γίνονται `export` στο shell του run. Εδώ μπαίνουν τα credentials των providers, όπως το `AWS_ACCESS_KEY_ID` του κεφαλαίου 17, ή ρυθμίσεις όπως το `TF_LOG`.

Κάθε μεταβλητή μπορεί να σημειωθεί **sensitive**. Τότε γίνεται write only και κανείς δεν ξαναβλέπει την τιμή, ούτε στο UI ούτε από το API. Το state όμως δεν προστατεύεται. Αν η configuration βάλει την τιμή σε output ή attribute, θα βρεθεί εκεί, όπως είδες στο κεφάλαιο 24.

Τα **variable sets** είναι συλλογές μεταβλητών για πολλά workspaces. Ένα set εφαρμόζεται **global**, σε όλα τα τωρινά και μελλοντικά workspaces του organization, σε **projects** ή σε επιλεγμένα **workspaces**. Ένα set με τα credentials του AWS στο project `web` καλύπτει κάθε νέο workspace του project.

Όταν το ίδιο κλειδί έρχεται από πολλές πηγές, κερδίζει η πιο ειδική. Από την ισχυρότερη στην ασθενέστερη η σειρά είναι η εξής.

1. Variable sets με σημάδι **priority**. Νικούν ακόμη και τη γραμμή εντολών.
2. `-var` και `-var-file` σε ένα CLI-driven run.
3. Τοπικές μεταβλητές `TF_VAR_` σε ένα CLI-driven run.
4. Μεταβλητές του ίδιου του workspace.
5. Variable sets εφαρμοσμένα σε workspaces ή σε projects.
6. Global variable sets.
7. Τα `*.auto.tfvars` και μετά το `terraform.tfvars` της configuration.

Ανάμεσα σε variable sets με το ίδιο κλειδί και το ίδιο εύρος κερδίζει το πρώτο αλφαβητικά. Το `A_credentials` νικά το `B_credentials`.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Η μεταβλητή του workspace νικά κάθε variable set χωρίς priority. Το global set είναι το ασθενέστερο από τα sets. Το priority set νικά τα πάντα. Σε local execution mode οι μεταβλητές του workspace αγνοούνται.

Το state στο HCP Terraform

Κάθε workspace έχει δικό του state, κρυπτογραφημένο στην αποθήκευση. Κάθε run που αλλάζει την υποδομή γράφει **νέα έκδοση του state**. Η καρτέλα States δείχνει κάθε έκδοση με το run, το commit και τη διαφορά από την προηγούμενη. Αν κάτι χαλάσει, κλειδώνεις το workspace και κάνεις **rollback** από το UI. Η παλιά έκδοση αντιγράφεται ως νέα τρέχουσα, οπότε τίποτα δεν χάνεται.

Το **locking** είναι αυτόματο. Όσο τρέχει ένα run, κανένα άλλο δεν ξεκινά. Κλειδώνεις επίσης ένα workspace με το χέρι για συντήρηση. Τα δικαιώματα ξεχωρίζουν ανάγνωση μόνο των outputs, ανάγνωση όλου του state και εγγραφή.

Outputs από άλλο workspace

Το `pets-prod` θέλει το subnet που δίνει ως output το `network-prod`. Ο πρώτος τρόπος είναι το `terraform_remote_state` του κεφαλαίου 19 με backend `remote`.

```hcl
data "terraform_remote_state" "network" {
  backend = "remote"

  config = {
    organization = "acme"
    workspaces = {
      name = "network-prod"
    }
  }
}
```

Τότε το `network-prod` πρέπει να επιτρέπει την ανάγνωση με τη ρύθμιση **remote state sharing**. Μοιράζεται με όλο το organization, με το project του ή με συγκεκριμένα workspaces. Ένα νέο workspace δεν μοιράζεται με κανένα.

Ο δεύτερος τρόπος, που προτείνει η HashiCorp, είναι το data source `tfe_outputs` του provider `hashicorp/tfe`. Διαβάζει μόνο outputs, χωρίς πρόσβαση σε όλο το state.

```hcl
data "tfe_outputs" "network" {
  organization = "acme"
  workspace    = "network-prod"
}

resource "terraform_data" "server" {
  input = data.tfe_outputs.network.values.subnet_id
}
```

Το `values` είναι ολόκληρο sensitive. Το `nonsensitive_values` περιέχει μόνο τα outputs που δεν ήταν sensitive. Ο provider `tfe` θέλει token, που σε remote run δίνεις με τη sensitive environment variable `TFE_TOKEN`.

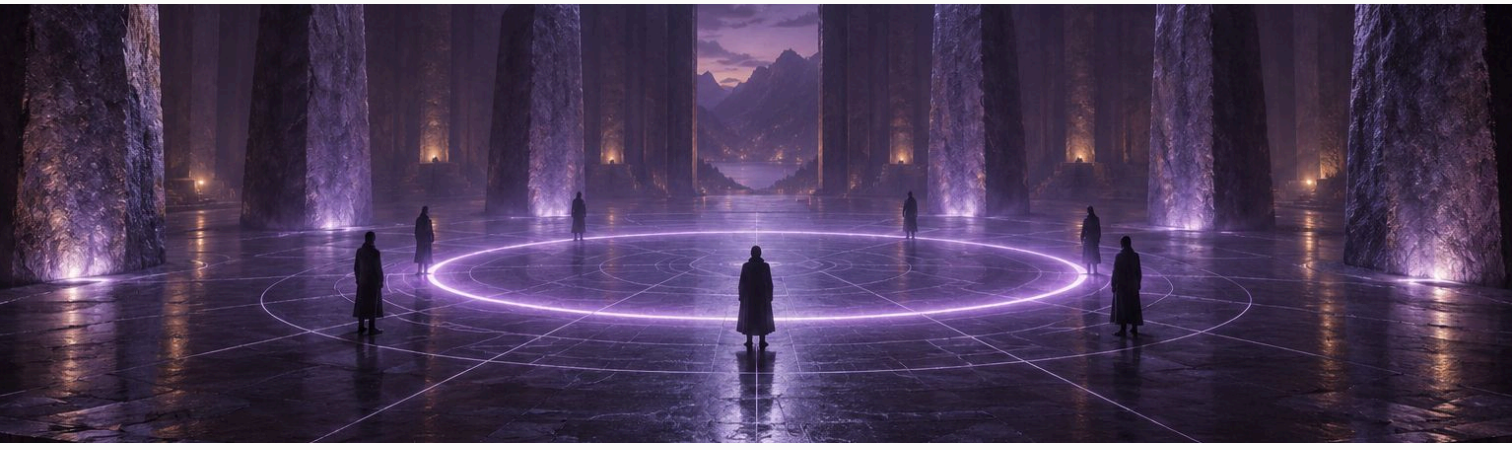
Όταν αλλάξει το `network-prod`, το `pets-prod` πρέπει να ξανατρέξει. Αυτό το κάνει ένα **run trigger**. Στο `pets-prod` δηλώνεις το `network-prod` ως source workspace και κάθε επιτυχημένο apply του βάζει run στην ουρά του `pets-prod`. Ένα workspace δέχεται έως 20 source workspaces. Τα runs από run trigger δεν κάνουν auto-apply, εκτός αν το ενεργοποιήσεις χωριστά.

Έλεγχος και διακυβέρνηση

Policies με Sentinel και OPA

Μια **policy** είναι κανόνας σε κώδικα που ελέγχει το plan πριν από το apply, για παράδειγμα ότι το production μένει σε συγκεκριμένα regions. Αυτό λέγεται policy as code. Το HCP Terraform υποστηρίζει δύο γλώσσες. Το **Sentinel** είναι της HashiCorp. Το **OPA**, το Open Policy Agent, γράφεται σε Rego.

Οι πολιτικές μπαίνουν σε **policy sets**. Ένα set έχει πολιτικές μίας γλώσσας και εφαρμόζεται σε όλο το organization, σε projects, σε workspaces ή σε workspaces με συγκεκριμένα tags. Συνήθως ζει σε



repository Git, ώστε κάθε αλλαγή του να περνά από review. Κάθε πολιτική έχει ένα **enforcement level**.

Sentinel	OPA	Όταν η πολιτική αποτύχει
`advisory`	`advisory`	Προειδοποίηση. Το run συνεχίζει.
`soft-mandatory`	`mandatory`	Το run σταματά, αλλά επιτρέπεται override.
`hard-mandatory`		Το run σταματά. Διορθώνεις τον κώδικα.

Το override το κάνει χρήστης με το σχετικό δικαίωμα. Ένα policy set μπορεί να επιτρέψει override ακόμη και στο hard-mandatory. Οι πολιτικές τρέχουν και στα speculative plans. Στο CLI φαίνονται μόνο τα αποτελέσματα του Sentinel.

ΕΝΔΕΙΚΤΙΚΗ ΕΞΟΔΟΣ · ΔΕΝ ΕΚΤΕΛΕΣΤΗΚΕ

```
Plan: 1 to add, 0 to change, 1 to destroy.

------------------------------------------------------------------------

Organization policy check:

Sentinel Result: false

Sentinel evaluated to false because one or more Sentinel policies evaluated
to false. This false was not due to an undefined value or runtime error.

1 policies evaluated.

## Policy 1: allowed-regions.sentinel (soft-mandatory)

Result: false

FALSE - allowed-regions.sentinel:1:1 - Rule "main"

Do you want to override the soft failed policy check?
  Only 'override' will be accepted to override.

  Enter a value: override
```

Μετά το plan απέτυχε η πολιτική `allowed-regions`. Είναι soft-mandatory, οπότε το Terraform ρωτά για override και δέχεται μόνο τη λέξη `override`. Με hard-mandatory δεν θα υπήρχε ερώτηση.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Μάθε τα τρία levels του Sentinel. Το advisory προειδοποιεί, το soft-mandatory σταματά αλλά παρακάμπτεται, το hard-mandatory σταματά. Οι πολιτικές τρέχουν μετά το plan και πριν από το apply.

ΣΗΜΕΙΩΣΗ

Το HCP Terraform δοκιμάζει και πολιτικές σε HCL, το Terraform policy, σε beta. Αυτό και τα Stacks, που θα δεις δίπλα στα workspaces, είναι εκτός ύλης.

Run tasks και cost estimation

Τα **run tasks** στέλνουν τα δεδομένα του run σε εξωτερική υπηρεσία σε τέσσερα σημεία, pre-plan, post-plan, pre-apply και post-apply. Η υπηρεσία απαντά pass ή fail. Ένα run task είναι advisory ή mandatory και ένα mandatory που αποτυγχάνει σταματά το run. Έτσι συνδέεις scanners ασφαλείας ή το HCP Packer, που ελέγχει αν τα images έχουν ανακληθεί.

Το **cost estimation** υπολογίζει το μηνιαίο κόστος των resources του AWS, του Azure και του Google Cloud και τη διαφορά που φέρνει το plan. Είναι κλειστό από προεπιλογή και τρέχει ως στάδιο ανάμεσα στο plan και στο apply. Μια Sentinel πολιτική μπορεί να σταματήσει μια ακριβή αλλαγή.

Health assessments

Τα **health assessments** ελέγχουν περιοδικά ένα workspace χωρίς να αλλάξουν τίποτα.

- **Drift detection.** Συγκρίνει την πραγματική υποδομή με την configuration, όπως έκανες με το χέρι στο κεφάλαιο 21. Αν κάποιος άλλαξε ένα bucket από την κονσόλα, το workspace δείχνει τι απέκλινε. Είτε κάνεις apply για να επαναφέρεις την υποδομή είτε αλλάζεις τον κώδικα.
- **Continuous validation.** Ξαναελέγχει τα preconditions, τα postconditions και κυρίως τα check blocks του κεφαλαίου 13. Ένα check που ρωτά αν ένα site απαντά με 200 γίνεται συνεχής παρακολούθηση.

Θέλουν remote ή agent execution και ένα επιτυχημένο apply. Ενεργοποιούνται ανά workspace ή επιβάλλονται σε όλο το organization και στέλνουν notifications.

Explorer και change requests

Ο **explorer** δείχνει όλο το organization σε πίνακες, για παράδειγμα ποια workspaces έχουν παλιά έκδοση ενός module ή drift. Από εκεί ένας διαχειριστής φτιάχνει **change requests**, εργασίες γραμμένες πάνω σε workspaces, όπως «αναβάθμισε το module `vpc`».

Private registry και no-code modules

Κάθε organization έχει ένα **private registry**, όπως είδες στο κεφάλαιο 16, με modules και providers ορατά μόνο στα μέλη του. Ένα private module έχει τέσσερα μέρη, `<HOSTNAME>/<ORGANIZATION>/<NAME>/<PROVIDER>`. Ένας private provider έχει τρία.

```hcl
terraform {
  required_providers {
    internal = {
      source  = "app.terraform.io/acme/internal"
      version = "1.4.0"
    }
  }
}

module "vpc" {
  source  = "app.terraform.io/acme/vpc/aws"
  version = "~> 2.3"

  cidr_block = "10.0.0.0/16"
}
```

Στο Terraform Enterprise το hostname είναι αυτό της εγκατάστασής σου. Σε remote run τα private modules κατεβαίνουν χωρίς τίποτα επιπλέον. Από το CLI το `terraform init` θέλει token από το `terraform login` ή το `TF_TOKEN_`.

Ένα module δημοσιεύεται από repository Git με τη δομή του κεφαλαίου 16. Με δημοσίευση βάσει **tags** κάθε Git tag με semantic version, όπως `v2.3.0`, γίνεται νέα έκδοση. Με δημοσίευση βάσει **branch** δηλώνεις κλαδί και αριθμό έκδοσης. Όσοι χρησιμοποιούν το module δεν χρειάζονται πρόσβαση στο repository.

Τα **no-code ready modules** απευθύνονται σε όσους δεν γράφουν Terraform. Διαλέγουν το module, συμπληρώνουν τις μεταβλητές σε φόρμα και το HCP Terraform φτιάχνει νέο workspace και κάνει apply. Ένα τέτοιο module δηλώνει μόνο του τους providers του. Τα credentials έρχονται συνήθως από variable set του project.

Ομάδες, δικαιώματα και ασφάλεια

Τα δικαιώματα δίνονται σε **teams**, σε τρία επίπεδα, organization, project και workspace. Αθροίζονται, οπότε ισχύει το υψηλότερο. Σε ένα workspace μια team παίρνει έναν ρόλο ή δικό σου σύνολο δικαιωμάτων.

- **Read.** Βλέπει runs, μεταβλητές και state.
- **Plan.** Ξεκινά επιπλέον plans.
- **Write.** Κάνει apply, γράφει μεταβλητές και state, κλειδώνει το workspace.
- **Admin.** Αλλάζει ρυθμίσεις και δικαιώματα teams και σβήνει το workspace.

Δώσε σε κάθε team μόνο ό,τι χρειάζεται. Σε workspace με VCS όποιος κάνει merge στο κλαδί ξεκινά runs έμμεσα, ακόμη κι αν δεν έχει δικαίωμα στο HCP Terraform.

Το organization συνδέεται με **SSO** μέσω SAML, για παράδειγμα με Okta ή Microsoft Entra ID. Με ενεργό SSO όλοι εκτός των owners μπαίνουν μέσω του identity provider. Τα **audit trails** είναι ροή γεγονότων από το API για το ποιος άλλαξε τι, με ιστορικό 14 ημερών.

Dynamic provider credentials

Ένα access key σε sensitive μεταβλητή δεν φαίνεται, αλλά ζει μέχρι να το ανακαλέσει κάποιος. Με τα **dynamic provider credentials** κάθε run παίρνει προσωρινά credentials μέσω **OIDC**, του OpenID Connect.

1. Μία φορά στήνεις σχέση εμπιστοσύνης. Στο AWS φτιάχνεις OIDC identity provider για το `https://app.terraform.io` και ένα IAM role που τον εμπιστεύεται, με όρους για organization, project, workspace και φάση.
2. Σε κάθε plan και apply το HCP Terraform φτιάχνει υπογεγραμμένο token ταυτότητας, για παράδειγμα με subject `organization:acme:project:web:workspace:pets-prod:run_phase:apply`.
3. Το AWS ελέγχει το token και δίνει προσωρινά credentials για το role, που πετιούνται στο τέλος του run.

Στο workspace ή σε variable set βάζεις το `TFC_AWS_PROVIDER_AUTH` με τιμή `true` και το `TFC_AWS_RUN_ROLE_ARN` με το ARN του role. Το block `provider "aws"` δεν έχει κλειδιά. Αντίστοιχος μηχανισμός υπάρχει για Azure, Google Cloud, Kubernetes και Vault.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Τα dynamic provider credentials είναι η προτεινόμενη λύση. Δεν υπάρχει μακρόβιο κλειδί για να διαρρεύσει ή να ανανεωθεί και τα δικαιώματα περιορίζονται ανά workspace και ανά φάση του run.

Ασκήσεις

Οι ασκήσεις αυτού του κεφαλαίου δεν έχουν αναμενόμενο αποτέλεσμα, γιατί θέλουν λογαριασμό. Γράψε τον κώδικα σε νέο φάκελο μέσα στο `~/tfbook/ch26`, έλεγξέ τον με `terraform fmt -check` και σύγκρινέ τον με τη λύση του τόμου των λύσεων. Με δωρεάν λογαριασμό μπορείς να τις τρέξεις, βάζοντας το δικό σου organization στη θέση του `acme`.

ΑΣΚΗΣΗ 26.1 Ένα συγκεκριμένο workspace

Το organization σου στο HCP Terraform λέγεται `acme`. Ο φάκελος `pets` πρέπει να δουλεύει με το workspace `pets-prod` και μόνο με αυτό. Γράψε το block `terraform` του `main.tf`, με το block `cloud` και τον provider `random` στην έκδοση 3.9.1. Η configuration πρέπει να δουλεύει στο `app.terraform.io`.

ΑΣΚΗΣΗ 26.2 Τρία περιβάλλοντα με tags

Τα workspaces `pets-dev` και `pets-prod` υπάρχουν ήδη στο project `web` του organization `acme` και έχουν το tag `app` με τιμή `pets`. Ο φάκελος `pets` πρέπει να εξυπηρετεί αυτά και ένα νέο `pets-staging`, ενώ οι προγραμματιστές αλλάζουν περιβάλλον με `terraform workspace select`. Γράψε το block `terraform` με το block `cloud`. Γράψε μετά τις εντολές που κάνουν init, φτιάχνουν το `pets-staging` από το CLI και δείχνουν τη λίστα των workspaces.

ΑΣΚΗΣΗ 26.3 Διόρθωσε το block cloud

Μια συνάδελφος θέλει να μεταφέρει τη configuration στο workspace `pets-dev` του organization `acme`. Έγραψε το παρακάτω `main.tf`, αλλά το `terraform init` αποτυγχάνει. Βρες τα τρία λάθη και γράψε το σωστό `main.tf`.

main.tf

```
variable "org" {
  type    = string
  default = "acme"
}

terraform {
  backend "s3" {
    bucket       = "acme-tfstate"
    key          = "pets/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }

  cloud {
    organization = var.org

    workspaces {
      name = "pets-dev"
      tags = ["pets"]
    }
  }
}
```

ΑΣΚΗΣΗ 26.4 Από το private registry

Το private registry του organization `acme` έχει το module `vpc` για τον provider `aws`, με εκδόσεις από 2.0.0 έως 2.3.1. Έχει επίσης τον δικό του provider `internal` στην έκδοση 1.4.0. Γράψε `main.tf` που δηλώνει τον provider με τοπικό όνομα `internal` και ακριβή έκδοση 1.4.0. Το ίδιο αρχείο καλεί το module με όνομα `network` και δέχεται κάθε έκδοση από την 2.3.0 και πάνω, αλλά όχι την 3.0.0. Δώσε στο module την τιμή `10.20.0.0/16` στο input `cidr_block`.

ΑΣΚΗΣΗ 26.5 Outputs από το network-prod

Το workspace `network-prod` του organization `acme` έχει το output `subnet_id`, που δεν είναι sensitive. Στο workspace `pets-prod` θέλεις ένα resource `terraform_data` με όνομα `server` και `input` την τιμή του `subnet_id`. Γράψε το data source `tfe_outputs` και το resource, ώστε η τιμή να μη γίνει sensitive. Ποια environment variable χρειάζεται το `pets-prod` για να δουλέψει ο provider `tfe` σε remote run;

ΑΣΚΗΣΗ 26.6 Από το S3 στο HCP Terraform

Ο φάκελος `pets` έχει state στο S3, με ένα block σαν αυτά του κεφαλαίου 19.

main.tf

```
terraform {
  backend "s3" {
    bucket       = "acme-tfstate"
    key          = "pets/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }

  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.9.1"
    }
  }
}
```

Μετάφερε το state στο workspace `pets-prod` του project `web` στο organization `acme`. Γράψε το νέο block `terraform` και την εντολή που κάνει τη μεταφορά. Τι απαντάς στην ερώτηση του Terraform;

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 26.1

Μια ομάδα χρησιμοποιεί CLI workspaces με backend S3 για τα περιβάλλοντα dev και prod. Σκέφτεται να περάσει στα workspaces του HCP Terraform. Ποια διαφορά ισχύει;

a. Τα workspaces του HCP Terraform μοιράζονται υποχρεωτικά τις ίδιες μεταβλητές.
b. Τα CLI workspaces έχουν το καθένα δικά του δικαιώματα πρόσβασης.
c. Κάθε workspace του HCP Terraform έχει δικές του μεταβλητές, δικό του state, δικά του δικαιώματα και δικό του ιστορικό runs.
d. Τα workspaces του HCP Terraform είναι προαιρετικά, όπως τα CLI workspaces.

ΕΡΩΤΗΣΗ 26.2

Ένα workspace είναι συνδεδεμένο με το κλαδί `main` ενός repository στο GitHub. Ένας προγραμματιστής ανοίγει pull request προς το `main`. Τι κάνει το HCP Terraform;

a. Κάνει plan και apply με τον κώδικα του pull request.
b. Τρέχει speculative plan και δείχνει το αποτέλεσμα μέσα στο pull request.
c. Τίποτα, μέχρι να γίνει merge.
d. Κλειδώνει το workspace μέχρι να κλείσει το pull request.

ΕΡΩΤΗΣΗ 26.3

Ποιο enforcement level του Sentinel σταματά το run όταν η πολιτική αποτύχει, αλλά επιτρέπει σε εξουσιοδοτημένο χρήστη να το συνεχίσει;

a. `advisory`
b. `hard-mandatory`
c. `soft-mandatory`
d. `mandatory`

ΕΡΩΤΗΣΗ 26.4

Τρέχεις `terraform login` χωρίς όρισμα. Πού αποθηκεύεται το token;

a. Σε απλό κείμενο στο `~/.terraform.d/credentials.tfrc.json`
b. Στο block `cloud` της configuration
c. Κρυπτογραφημένο στο `terraform.tfstate`
d. Στη μεταβλητή περιβάλλοντος `TF_TOKEN_app_terraform_io`

ΕΡΩΤΗΣΗ 26.5

Ποιοι δύο τρόποι δίνουν το όνομα του organization στο block `cloud`; Επίλεξε δύο.

a. Το argument `organization` μέσα στο block
b. Η μεταβλητή περιβάλλοντος `TF_CLOUD_ORGANIZATION`
c. Μια input variable, για παράδειγμα `organization = var.org`
d. Ένα local, για παράδειγμα `organization = local.org`

ΕΡΩΤΗΣΗ 26.6

Ένα workspace έχει τη Terraform variable `region` με τιμή `eu-central-1`. Ένα global variable set, χωρίς priority, ορίζει `region` με τιμή `us-east-1`. Κανείς δεν δίνει `-var`. Ποια τιμή χρησιμοποιεί το run;

a. `us-east-1`, γιατί τα variable sets νικούν τις μεταβλητές του workspace.
b. Αυτή που άλλαξε τελευταία.
c. Το run αποτυγχάνει λόγω σύγκρουσης.
d. `eu-central-1`, γιατί η μεταβλητή του workspace νικά τα variable sets.

ΕΡΩΤΗΣΗ 26.7

Ένα workspace έχει execution mode Local. Τι κάνει το HCP Terraform για αυτό;

a. Τρέχει plan και apply στα δικά του μηχανήματα και στέλνει την έξοδο στο CLI.
b. Αποθηκεύει και κλειδώνει το state. Το plan και το apply τρέχουν στο μηχάνημά σου.
c. Εφαρμόζει τις μεταβλητές του workspace στο τοπικό run.
d. Τρέχει τις Sentinel πολιτικές πριν από κάθε τοπικό apply.

ΕΡΩΤΗΣΗ 26.8

Σωστό ή λάθος; Σε workspace συνδεδεμένο με repository VCS μπορείς να τρέξεις `terraform plan` από το CLI, αλλά όχι `terraform apply`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 26.9

Η εταιρεία θέλει το HCP Terraform να διαχειρίζεται servers σε ιδιωτικό datacenter. Το δίκτυο δεν επιτρέπει καμία εισερχόμενη σύνδεση από το internet. Ποια λύση ταιριάζει;

a. Execution mode Local σε όλα τα workspaces
b. Ένα run task σε φάση pre-plan
c. HCP Terraform agents μέσα στο datacenter
d. Dynamic provider credentials

ΕΡΩΤΗΣΗ 26.10

Ποιο είναι το κύριο όφελος των dynamic provider credentials έναντι ενός access key σε sensitive μεταβλητή του workspace;

a. Το access key δεν εμφανίζεται πια στα logs.
b. Δεν χρειάζεται IAM role στο AWS.
c. Το plan τρέχει πιο γρήγορα.
d. Κάθε run παίρνει προσωρινά credentials μέσω OIDC και δεν υπάρχει μακρόβιο κλειδί για ανανέωση ή διαρροή.

ΕΡΩΤΗΣΗ 26.11

Τι κάνει η λειτουργία continuous validation των health assessments;

a. Τρέχει apply κάθε 24 ώρες για να διορθώσει το drift.
b. Ελέγχει περιοδικά τα check blocks, τα preconditions και τα postconditions της configuration χωρίς να αλλάξει την υποδομή.
c. Κάνει `terraform validate` σε κάθε commit.
d. Συγκρίνει το κόστος της υποδομής με τον προϋπολογισμό.

ΕΡΩΤΗΣΗ 26.12

Ποια είναι η σωστή τιμή του `source` για το module `vpc` του provider `aws`, δημοσιευμένο στο private registry του organization `acme` στο HCP Terraform;

a. `acme/vpc/aws`
b. `app.terraform.io/vpc/aws`
c. `app.terraform.io/acme/vpc/aws`
d. `registry.terraform.io/acme/vpc`



ΚΕΦΑΛΑΙΟ 27

Τελικό project

Σε αυτό το κεφάλαιο χτίζεις από το μηδέν ένα μικρό αλλά πραγματικό project στο AWS. Remote state με lock, ένα module με `for_each`, validation, outputs, data source και check. Μετά ζεις τρεις αλλαγές που συμβαίνουν σε κάθε project που μένει ζωντανό. Ένα refactoring, ένα drift και ένα import. Στο τέλος τα σβήνεις όλα και βεβαιώνεσαι ότι δεν έμεινε τίποτα. Κάθε βήμα στηρίζεται σε κεφάλαιο που έχεις ήδη διαβάσει.

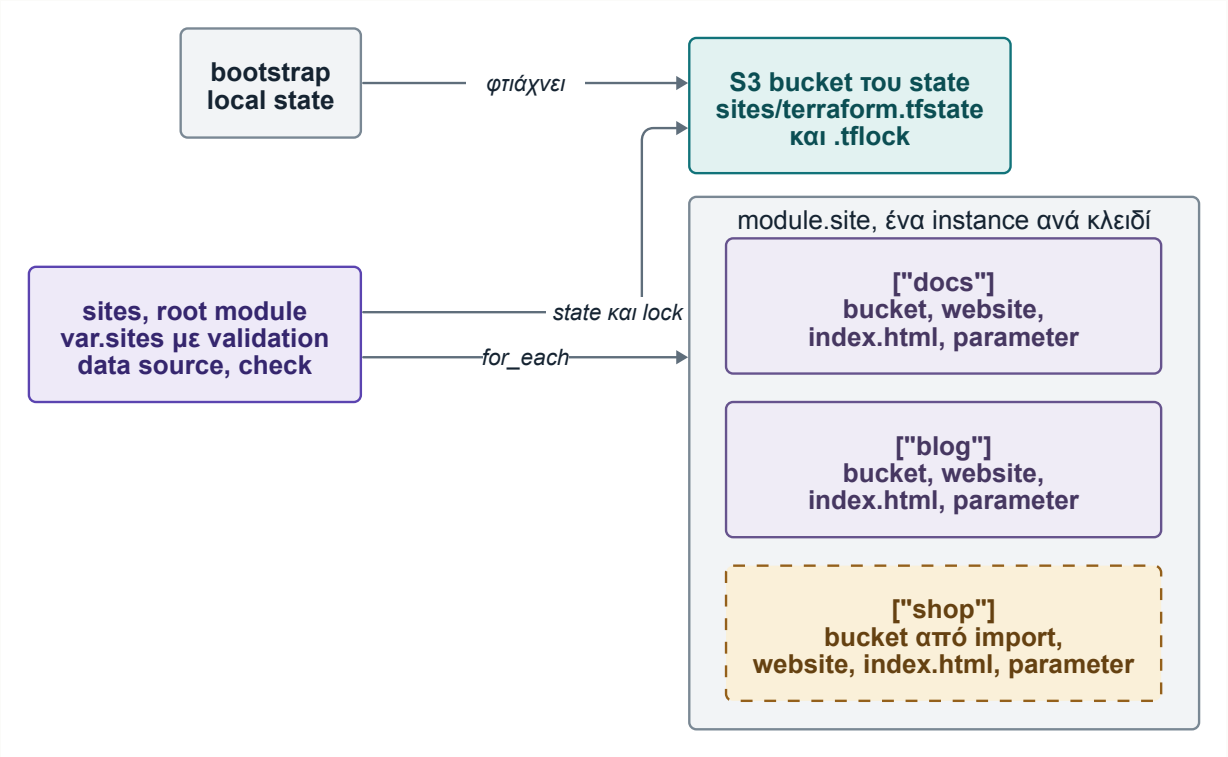
Τι θα χτίσεις

Μια μικρή ομάδα φιλοξενεί στατικά sites στο S3. Την τεκμηρίωση, το blog και αργότερα το κατάστημα. Ως τώρα κάθε bucket φτιαχνόταν με το χέρι. Η ομάδα θέλει να περάσει τα sites στο Terraform και γράφει πέντε απαιτήσεις.

1. Το state ζει σε S3 bucket με lock, ώστε να το μοιράζονται όλοι.
2. Ένα νέο site είναι μία γραμμή σε μια μεταβλητή. Ένα module φτιάχνει για κάθε site το bucket, τη ρύθμιση website, την αρχική σελίδα και ένα parameter με το όνομα του bucket.
3. Ένα λάθος όνομα site σταματά στο plan και δεν φτάνει ποτέ στο AWS.
4. Το pipeline που ανεβάζει τις σελίδες βρίσκει το bucket κάθε site στο SSM Parameter Store.
5. Κάθε plan προειδοποιεί αν στο Parameter Store υπάρχουν parameters που δεν ανήκουν σε κανένα site.

Δύο υπηρεσίες είναι καινούργιες για το βιβλίο. Το S3 μπορεί να σερβίρει τα αρχεία ενός bucket ως website. Το resource `aws_s3_bucket_website_configuration` λέει ποιο αρχείο είναι η αρχική σελίδα και δίνει στο bucket μια διεύθυνση website. Το **SSM Parameter Store** κρατά ζεύγη ονόματος και τιμής σε μια ιεραρχία που μοιάζει με φακέλους, για παράδειγμα `/sites/docs/bucket`. Ένα script το διαβάζει με το `aws`

CLI χωρίς να ξέρει τίποτα για το Terraform. Κάθε τέτοιο ζεύγος το φτιάχνει ένα resource `aws_ssm_parameter`.



Σχήμα 27.1 · Το project έχει δύο configurations. Το bootstrap φτιάχνει μόνο το bucket του state. Το sites κρατά εκεί το state του και φτιάχνει ένα instance του module για κάθε κλειδί της μεταβλητής sites.

Το project έχει δύο φακέλους. Το `bootstrap` φτιάχνει μόνο το bucket του state και κρατά το δικό του state τοπικά. Το `sites` είναι η κύρια configuration. Καλεί το module `site` μία φορά για κάθε site και κρατά το state του στο bucket του `bootstrap`.

Βήμα 1, το bucket του state

Το κεφάλαιο 19 εξήγησε το πρόβλημα της κότας και του αυγού. Μια configuration δεν μπορεί να κρατά το state της σε bucket που φτιάχνει η ίδια, γιατί το χρειάζεται ήδη στο πρώτο `init`. Γι' αυτό το bucket έχει δική του μικρή configuration με local state.

TERMINAL

```
mkdir bootstrap
cd bootstrap
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
```

```
      version = "6.67.0"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

data "aws_caller_identity" "current" {}

resource "aws_s3_bucket" "state" {
  bucket = "tfbook-${data.aws_caller_identity.current.account_id}-state"
}

output "bucket" {
  value = aws_s3_bucket.state.bucket
}
```

Το όνομα του bucket περιέχει τον αριθμό του λογαριασμού, όπως στο κεφάλαιο 17. Έτσι είναι μοναδικό και στο πραγματικό AWS. Το output `bucket` θα το χρειαστεί η επόμενη configuration.

TERMINAL

```
terraform init
```

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
aws_s3_bucket.state: Creating...
aws_s3_bucket.state: Creation complete after 1s [id=tfbook-123456789012-state]
…
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
…
bucket = "tfbook-123456789012-state"
```

Το bucket φτιάχτηκε και το output δίνει το όνομά του. Το state αυτής της configuration είναι το `terraform.tfstate` του φακέλου. Είναι μικρό και αλλάζει σπάνια. Μερικές ομάδες, όταν υπάρξει το bucket, το μεταφέρουν κι αυτό μέσα του με το `init -migrate-state` του κεφαλαίου 19.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Στο πραγματικό AWS το bucket του state παίρνει και `aws_s3_bucket_versioning`, ώστε να μπορείς να γυρίσεις σε παλιό state. Παίρνει επίσης `prevent_destroy` από το κεφάλαιο 14. Εδώ λείπουν για έναν λόγο. Στο τέλος του κεφαλαίου σβήνεις τα πάντα. Το Moto 5.2.3 όμως δεν μπορεί να σβήσει τις παλιές εκδόσεις που αφήνει το `.tflock` του state σε bucket με versioning. Η διαγραφή του bucket θα κολλούσε. Σε δικό σου λογαριασμό βάλε versioning και θυμήσου ότι ένα bucket με versioning σβήνεται μόνο αφού σβήσεις όλες τις εκδόσεις του.

Βήμα 2, το module site

Φτιάξε τον φάκελο της κύριας configuration. Αντέγραψε το lock file του `bootstrap`, όπως στο κεφάλαιο 17, ώστε το `init` να πάρει τον provider από το cache και να μην τον κατεβάσει ξανά.

TERMINAL

```
cd ..
mkdir sites
cd sites
cp ../bootstrap/.terraform.lock.hcl .
```

Το module ζει στο `modules/site` και τα αρχεία του χωρίζονται ανά ρόλο, όπως στο κεφάλαιο 15. Το `versions.tf` δηλώνει τον provider με ελάχιστη έκδοση. Την ακριβή την καρφώνει το root module.

modules/site/versions.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.67.0"
    }
  }
}
```

Οι τρεις μεταβλητές είναι υποχρεωτικές. Το `name` είναι το σύντομο όνομα, το `bucket` το πλήρες όνομα του bucket και το `owner` ο υπεύθυνος.

modules/site/variables.tf

```
variable "name" {
  type        = string
  description = "Το σύντομο όνομα του site, για παράδειγμα docs"
}

variable "bucket" {
  type        = string
  description = "Το πλήρες όνομα του S3 bucket"
}

variable "owner" {
  type        = string
  description = "Ποιος έχει την ευθύνη του site"
}
```

modules/site/main.tf

```
resource "aws_s3_bucket" "this" {
  bucket = var.bucket

  tags = {
    Site  = var.name
    Owner = var.owner
  }
}
```

```
resource "aws_s3_bucket_website_configuration" "this" {
  bucket = aws_s3_bucket.this.id

  index_document {
    suffix = "index.html"
  }
}

resource "aws_s3_object" "index" {
  bucket       = aws_s3_bucket.this.id
  key          = "index.html"
  content      = "<h1>${var.name}</h1>\n"
  content_type = "text/html"
}
```

Τρία resources ανά site. Το bucket παίρνει δύο tags από τις μεταβλητές. Η ρύθμιση website και η αρχική σελίδα αναφέρονται στο `aws_s3_bucket.this.id`, οπότε φτιάχνονται μετά το bucket, όπως έμαθες για τις implicit dependencies στο κεφάλαιο 6. Το module δεν έχει block `provider`. Παίρνει τον provider του root module.

modules/site/outputs.tf

```
output "bucket" {
  value = aws_s3_bucket.this.bucket
}

output "endpoint" {
  value = aws_s3_bucket_website_configuration.this.website_endpoint
}
```

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Στο AWS ένα bucket με ρύθμιση website δεν είναι ακόμη δημόσιο. Τα νέα buckets μπλοκάρουν τη δημόσια πρόσβαση, οπότε η διεύθυνση website απαντά με `403 Forbidden`. Για ένα πραγματικό site χρειάζεσαι bucket policy που επιτρέπει την ανάγνωση και χαλάρωση του block public access, συνήθως και CloudFront μπροστά. Το project σταματά πριν από αυτό, γιατί για να μάθεις Terraform δεν χρειάζεσαι δημόσιο bucket.



Βήμα 3, το root module

Το root module έχει περισσότερα αρχεία, το καθένα με έναν ρόλο. Ξεκίνα από τις εκδόσεις και το backend.

versions.tf

```
terraform {
  required_version = ">= 1.12"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.67.0"
    }
  }
}
```

backend.tf

```
terraform {
  backend "s3" {
    key          = "sites/terraform.tfstate"
    region       = "eu-central-1"
    use_lockfile = true
  }
}
```

Το `required_version` απορρίπτει εκδόσεις του Terraform παλιότερες από την 1.12 (κεφάλαιο 5). Το backend είναι partial configuration, όπως στο κεφάλαιο 19. Λείπει το `bucket`, γιατί το όνομά του έχει τον αριθμό του λογαριασμού και το block `backend` δέχεται μόνο σταθερές τιμές. Θα το δώσεις στο `init`. Το `use_lockfile` κλειδώνει το state με ένα αντικείμενο `.tflock` δίπλα του.

main.tf

```
provider "aws" {
  region = "eu-central-1"

  default_tags {
    tags = {
      Project   = "sites"
      ManagedBy = "terraform"
    }
  }
}

data "aws_caller_identity" "current" {}

module "site" {
  source   = "./modules/site"
  for_each = var.sites

  name   = each.key
  bucket = "tfbook-${data.aws_caller_identity.current.account_id}-${each.key}"
  owner  = each.value.owner
}
```

Τα `default_tags` του κεφαλαίου 17 πάνε σε κάθε resource που φτιάχνει ο provider. Το module έχει `for_each` πάνω στη μεταβλητή `sites`. Κάθε κλειδί γίνεται ένα instance, για παράδειγμα `module.site["docs"]`. Το `each.key` είναι το όνομα του site και το `each.value` το object με τα στοιχεία του. Το όνομα του bucket ενώνει τον λογαριασμό από το data source με το κλειδί.

Τα parameters μπαίνουν προς το παρόν στο root module, σε δικό τους αρχείο.

parameters.tf

```
resource "aws_ssm_parameter" "site" {
  for_each = module.site

  name  = "/sites/${each.key}/bucket"
  type  = "String"
  value = each.value.bucket
}
```

Ένα module με `for_each` είναι map από instances, με τα ίδια κλειδιά. Γι' αυτό δουλεύει ως τιμή για ένα δεύτερο `for_each`. Εδώ το `each.value` είναι ένα instance του module και το `each.value.bucket` το output του. Ο provider σημειώνει το `value` κάθε parameter ως sensitive, ακόμη και για τύπο `String`, οπότε τα plans το γράφουν `(sensitive value)` (κεφάλαιο 24).

variables.tf

```
variable "sites" {
  description = "Τα στατικά sites. Το κλειδί είναι το σύντομο όνομα του site."
  type = map(object({
    owner = string
  }))

  validation {
    condition     = alltrue([for name in keys(var.sites) : can(regex("^[a-z0-9]{3,12}$", name))])
    error_message = "Κάθε κλειδί του sites θέλει 3 έως 12 πεζά λατινικά γράμματα ή ψηφία."
  }
}
```

terraform.tfvars

```
sites = {
  docs = { owner = "maria" }
  blog = { owner = "nikos" }
}
```

Η μεταβλητή είναι `map(object(...))`, ο συνδυασμός τύπων του κεφαλαίου 9. Το κλειδί μπαίνει στο όνομα του bucket, που δέχεται μόνο πεζά γράμματα, ψηφία, τελείες και παύλες (κεφάλαιο 17). Η validation του κεφαλαίου 13 κρατά τα κλειδιά πιο αυστηρά. Η `regex` αποτυγχάνει όταν το κλειδί δεν ταιριάζει και η `can` κάνει την αποτυχία `false`. Η `alltrue` είναι νέα. Δέχεται λίστα από bool και δίνει `true` μόνο όταν όλα τα στοιχεία είναι `true`. Εδώ απαιτεί όλα τα κλειδιά να περνούν.

outputs.tf

```
output "buckets" {
  description = "Το bucket κάθε site"
  value       = { for name, site in module.site : name => site.bucket }
}

output "endpoints" {
  description = "Η διεύθυνση website κάθε site"
  value       = { for name, site in module.site : name => site.endpoint }
}
```

Τα outputs είναι `for` expressions του κεφαλαίου 10 πάνω στο map των instances. Κάθε ένα δίνει ένα map από όνομα site σε τιμή.

Τελευταίο είναι το check της απαίτησης 5.

checks.tf

```
data "aws_ssm_parameters_by_path" "sites" {
  path      = "/sites"
  recursive = true
}

check "parameters" {
  assert {
    condition = alltrue([
      for name in data.aws_ssm_parameters_by_path.sites.names :
      contains(keys(var.sites), split("/", name)[2])
    ])
    error_message = "Υπάρχουν parameters στο /sites χωρίς site στη μεταβλητή sites."
  }
}
```

Το data source διαβάζει όλα τα parameters κάτω από το `/sites`. Η `split` κόβει το `/sites/docs/bucket` στα κομμάτια `""`, `"sites"`, `"docs"` και `"bucket"`, οπότε το στοιχείο `[2]` είναι το όνομα του site. Το assert απαιτεί κάθε τέτοιο όνομα να υπάρχει στη μεταβλητή. Το data source μένει έξω από το check. Το κεφάλαιο 13 έδειξε ότι ένα scoped data source ξαναδιαβάζεται σε κάθε apply. Έτσι κάθε plan θα έδειχνε μια ενέργεια ανάγνωσης και δεν θα έλεγε ποτέ `No changes`. Εδώ θέλουμε καθαρά plans, για να φαίνεται το drift.

Τώρα το `init`. Το όνομα του bucket το δίνει το output του `bootstrap`.

TERMINAL

```
terraform init -backend-config="bucket=$(cd ../bootstrap && terraform output -raw bucket)"
```

ΕΞΟΔΟΣ

```
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing modules...
- site in modules/site
…
```

Το `$( )` του shell τρέχει το `terraform output -raw` στον φάκελο `bootstrap` και βάζει το όνομα στη θέση του. Το `-raw` του κεφαλαίου 8 υπάρχει ακριβώς για τέτοιες χρήσεις. Το `init` ρύθμισε το S3 backend και βρήκε το module `site` στο `modules/site`.

Πριν από το πρώτο plan έλεγξε μορφή και εγκυρότητα, όπως στο κεφάλαιο 2.

TERMINAL

```
terraform fmt -check -recursive
terraform validate
```

ΕΞΟΔΟΣ

```
Success! The configuration is valid.
```

Το `fmt -check` δεν τύπωσε τίποτα, άρα όλα τα αρχεία είναι ήδη σε μορφή. Δοκίμασε και τη validation με ένα κακό όνομα. Η τιμή με `-var` κερδίζει το `terraform.tfvars`, όπως είδες στο κεφάλαιο 7.

TERMINAL

```
terraform plan -var 'sites={ Eshop = { owner = "eleni" } }'
```

ΕΞΟΔΟΣ

```
…
╷
│ Error: Invalid value for variable
│
│   on variables.tf line 1:
│    1: variable "sites" {
│      ├────────────────
│      │ var.sites is map of object with 1 element
│
│ Κάθε κλειδί του sites θέλει 3 έως 12 πεζά λατινικά γράμματα ή ψηφία.
│
│ This was checked by the validation rule at variables.tf:7,3-13.
╵
```

Το plan σταμάτησε με το μήνυμά σου πριν σχεδιάσει οτιδήποτε. Το κεφαλαίο `E` δεν θα περνούσε τους κανόνες ονομάτων του S3. Χωρίς validation το λάθος θα φαινόταν μόνο στο apply, όταν το AWS θα απέρριπτε τη δημιουργία.

Βήμα 4, το πρώτο apply

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_ssm_parameter.site["blog"] will be created
…
  # aws_ssm_parameter.site["docs"] will be created
…
  # module.site["blog"].aws_s3_bucket.this will be created
…
  # module.site["blog"].aws_s3_bucket_website_configuration.this will be created
…
  # module.site["blog"].aws_s3_object.index will be created
…
  # module.site["docs"].aws_s3_bucket.this will be created
…
  # module.site["docs"].aws_s3_bucket_website_configuration.this will be created
…
  # module.site["docs"].aws_s3_object.index will be created
…
Plan: 8 to add, 0 to change, 0 to destroy.
…
```

Δύο sites επί τέσσερα resources, ένα bucket, μία ρύθμιση website, ένα object και ένα parameter, κάνουν `8 to add`. Οι διευθύνσεις μέσα στο module έχουν μπροστά το `module.site` και το κλειδί του instance. Τα parameters είναι ακόμη στο root, ως `aws_ssm_parameter.site`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

buckets = {
  "blog" = "tfbook-123456789012-blog"
  "docs" = "tfbook-123456789012-docs"
}
endpoints = {
  "blog" = "tfbook-123456789012-blog.s3-website.eu-central-1.amazonaws.com"
  "docs" = "tfbook-123456789012-docs.s3-website.eu-central-1.amazonaws.com"
}
```

Τα outputs δίνουν τα δύο buckets και τις διευθύνσεις website.

ΣΤΟ ΠΡΑΓΜΑΤΙΚΟ AWS

Ο provider στέλνει τα tags ενός νέου bucket μαζί με την αίτηση δημιουργίας. Το κεφάλαιο 17 έδειξε ότι το Moto 5.2.3 τα αγνοεί σε αυτή την αίτηση. Αν δουλεύεις με Moto, το επόμενο plan θα έδειχνε τα tags των δύο buckets ως update in-place. Γι' αυτό εδώ τρέχει ένα δεύτερο apply που τα γράφει. Στο AWS το ίδιο apply δεν βρίσκει τίποτα να κάνει.

TERMINAL

```
terraform apply -auto-approve
```

Έλεγξε πού κατέληξε το state και τι βλέπει το pipeline. Η μεταβλητή `ACCOUNT` κρατά τον αριθμό του λογαριασμού, όπως στο κεφάλαιο 17.

TERMINAL

```
ls
ACCOUNT=$(aws sts get-caller-identity --query Account --output text)
aws s3api list-objects-v2 --bucket "tfbook-$ACCOUNT-state" --query 'Contents[].Key'
aws ssm get-parameters-by-path --path /sites --recursive \
  --query 'Parameters[].[Name,Value]' --output text
```

ΕΞΟΔΟΣ

```
backend.tf
checks.tf
main.tf
modules
outputs.tf
parameters.tf
terraform.tfvars
variables.tf
versions.tf
[
    "sites/terraform.tfstate"
]
/sites/blog/bucket  tfbook-123456789012-blog
/sites/docs/bucket  tfbook-123456789012-docs
```

Στον φάκελο δεν υπάρχει `terraform.tfstate`. Το state είναι το αντικείμενο `sites/terraform.tfstate` στο bucket. Το `.tflock` δεν φαίνεται, γιατί υπάρχει μόνο όσο τρέχει μια εντολή. Τα δύο parameters δείχνουν το bucket κάθε site. Οι απαιτήσεις 1 έως 5 ισχύουν.

Τρεις αλλαγές στη ζωή του project

Ένα project δεν τελειώνει στο πρώτο apply. Ο κώδικας ξαναγράφεται, κάποιος αλλάζει κάτι από την κονσόλα και εμφανίζονται resources που φτιάχτηκαν πριν από το Terraform. Ακολουθούν οι τρεις περιπτώσεις, με τη σειρά που συμβαίνουν συνήθως.



Refactoring, το parameter μπαίνει στο module

Η ομάδα αποφασίζει ότι το parameter ανήκει στο site. Όποιος καλέσει το module σε άλλο project πρέπει να το παίρνει μαζί, χωρίς δεύτερο `for_each` στο root. Μετέφερε το resource στο module.

modules/site/parameter.tf

```
resource "aws_ssm_parameter" "bucket" {
  name  = "/sites/${var.name}/bucket"
  type  = "String"
  value = aws_s3_bucket.this.bucket
}
```

TERMINAL

```
rm parameters.tf
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_ssm_parameter.site["blog"] will be destroyed
  # (because aws_ssm_parameter.site is not in configuration)
…
  # aws_ssm_parameter.site["docs"] will be destroyed
  # (because aws_ssm_parameter.site is not in configuration)
…
  # module.site["blog"].aws_ssm_parameter.bucket will be created
…
  # module.site["docs"].aws_ssm_parameter.bucket will be created
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Το Terraform δεν ξέρει ότι πρόκειται για τα ίδια αντικείμενα. Βλέπει δύο διευθύνσεις που χάθηκαν από τη configuration και δύο καινούργιες, όπως εξήγησε το κεφάλαιο 23. Τα parameters θα σβήνονταν και θα ξαναφτιάχνονταν με νέο ιστορικό εκδόσεων. Το block `moved` λύνει το πρόβλημα. Ο πρώτος πειρασμός είναι ένα block για όλο το resource.

moved.tf

```
moved {
  from = aws_ssm_parameter.site
  to   = module.site.aws_ssm_parameter.bucket
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # module.site.aws_ssm_parameter.bucket["blog"] will be destroyed
  # (because module.site is not in configuration)
  # (moved from aws_ssm_parameter.site["blog"])
…
  # module.site.aws_ssm_parameter.bucket["docs"] will be destroyed
  # (because module.site is not in configuration)
  # (moved from aws_ssm_parameter.site["docs"])
…
  # module.site["blog"].aws_ssm_parameter.bucket will be created
…
  # module.site["docs"].aws_ssm_parameter.bucket will be created
…
Plan: 2 to add, 0 to change, 2 to destroy.
…
```

Η περίληψη δεν άλλαξε. Διάβασε τις γραμμές με `#`. Το Terraform εφάρμοσε το block, αλλά το `module.site` χωρίς κλειδί είναι άλλη διεύθυνση από το `module.site["docs"]`. Τα instances πήγαν στο `module.site.aws_ssm_parameter.bucket["docs"]`, όπου το κλειδί κόλλησε στο resource και όχι στο module. Αυτή η διεύθυνση δεν υπάρχει στη configuration, οπότε το plan τα σβήνει `because module.site is not in configuration`. Το `terraform validate` δεν θα το έπιανε. Μόνο το plan το δείχνει.

Ένα module με `for_each` δεν έχει instance χωρίς κλειδί. Γράψε ένα block για κάθε instance, με το κλειδί στο module.

moved.tf

```
moved {
  from = aws_ssm_parameter.site["docs"]
  to   = module.site["docs"].aws_ssm_parameter.bucket
}

moved {
  from = aws_ssm_parameter.site["blog"]
  to   = module.site["blog"].aws_ssm_parameter.bucket
}
```

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # aws_ssm_parameter.site["blog"] has moved to module.site["blog"].aws_ssm_parameter.bucket
…
  # aws_ssm_parameter.site["docs"] has moved to module.site["docs"].aws_ssm_parameter.bucket
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

Τώρα κάθε parameter γράφει `has moved to` με τη σωστή διεύθυνση και η περίληψη είναι `0 to add, 0 to change, 0 to destroy`. Αυτό είναι το αποτέλεσμα που ζητάς από κάθε refactoring.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
…
```

TERMINAL

```
terraform state list -id=/sites/docs/bucket
aws ssm get-parameter --name /sites/docs/bucket --query Parameter.Version
```

ΕΞΟΔΟΣ

```
module.site["docs"].aws_ssm_parameter.bucket
1
```

Το `state list -id` βρίσκει τη διεύθυνση από το ID του αντικειμένου. Το parameter ζει πια μέσα στο module. Η έκδοσή του είναι ακόμη `1`, άρα το AWS δεν το άγγιξε. Στο `sites` υπάρχει ένα state και πέρασε το apply, οπότε το `moved.tf` δεν χρειάζεται άλλο (κεφάλαιο 23).

TERMINAL

```
rm moved.tf
```

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Σε resource ή module με `for_each` κάθε instance έχει διεύθυνση με κλειδί σε αγκύλες. Το `module.site["docs"].aws_ssm_parameter.bucket` και το `module.site.aws_ssm_parameter.bucket["docs"]` είναι διαφορετικές διευθύνσεις. Η εξέταση τις βάζει δίπλα δίπλα στις επιλογές. Το ίδιο ισχύει στα `moved`, στα `import` και στις εντολές `state`. Το `moved` δεν δέχεται `for_each`, οπότε γράφεις ένα block ανά instance.

Drift, μια αλλαγή από την κονσόλα

Κάποιος από την ομάδα του blog αλλάζει από την κονσόλα την αρχική σελίδα σε `home.html`. Κάνε το ίδιο με το CLI.

TERMINAL

```
aws s3api put-bucket-website --bucket "tfbook-$ACCOUNT-blog" \
  --website-configuration '{"IndexDocument": {"Suffix": "home.html"}}'
```

Το κεφάλαιο 21 έδειξε πώς βλέπεις τι άλλαξε χωρίς να προτείνει το Terraform καμία ενέργεια.

TERMINAL

```
terraform plan -refresh-only
```

ΕΞΟΔΟΣ

```
…
  # module.site["blog"].aws_s3_bucket.this has changed
…
          ~ index_document           = "index.html" -> "home.html"
…
  # module.site["blog"].aws_s3_bucket_website_configuration.this has changed
…
          ~ suffix = "index.html" -> "home.html"
…
This is a refresh-only plan, so Terraform will not take any actions to undo
these. If you were expecting these changes then you can apply this plan to
record the updated values in the Terraform state without changing any remote
objects.
…
```

Το refresh-only plan συγκρίνει το state με τον πραγματικό κόσμο και όχι με τη configuration. Η ίδια αλλαγή φαίνεται σε δύο resources. Το `aws_s3_bucket_website_configuration` είναι αυτό που άλλαξε. Το `aws_s3_bucket` διαβάζει κι αυτό τη ρύθμιση website σε ένα attribute που μόνο διαβάζεται, άρα βλέπει την ίδια αλλαγή. Το κείμενο στο τέλος λέει ότι ένα refresh-only plan δεν αναιρεί τίποτα.

Τώρα αποφασίζεις. Αν η αλλαγή ήταν σωστή, τρέχεις `terraform apply -refresh-only`, που γράφει τις νέες τιμές στο state. Αλλάζεις επίσης τον κώδικα, αλλιώς το επόμενο κανονικό apply θα αναιρέσει την αλλαγή. Εδώ η αλλαγή ήταν λάθος, γιατί η σελίδα του blog είναι το `index.html`. Ο κώδικας είναι η πηγή της αλήθειας, οπότε ένα κανονικό apply την αναιρεί.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
  # module.site["blog"].aws_s3_bucket_website_configuration.this will be updated in-place
…
          ~ suffix = "home.html" -> "index.html"
…
Plan: 0 to add, 1 to change, 0 to destroy.
…
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
…
```

Το κανονικό plan δείχνει τη διόρθωση ως `~` update in-place στη ρύθμιση website, με το `suffix` από `home.html` πίσω σε `index.html`.

ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ

Το `terraform plan -refresh-only` δείχνει το drift χωρίς να προτείνει αλλαγές στην υποδομή. Το `terraform apply -refresh-only` το καταγράφει στο state και δεν αγγίζει κανένα αντικείμενο. Το `terraform refresh` κάνει το ίδιο χωρίς να δείχνει plan και χωρίς να ζητά επιβεβαίωση. Γι' αυτό θεωρείται παρωχημένο. Ένα κανονικό `terraform apply` κάνει το αντίθετο. Φέρνει την υποδομή πίσω στη configuration.

Import, το bucket του καταστήματος

Η ομάδα του καταστήματος είχε φτιάξει το bucket της με το χέρι, πριν υπάρξει το project. Δεν πρόλαβε να ανεβάσει τίποτα. Φτιάξε το με το CLI.

TERMINAL

```
aws s3 mb "s3://tfbook-$ACCOUNT-shop"
```

ΕΞΟΔΟΣ

```
make_bucket: tfbook-123456789012-shop
```

Το site μπαίνει στη μεταβλητή όπως κάθε άλλο. Αν σταματούσες εδώ, το plan θα σχεδίαζε νέο bucket με το ίδιο όνομα και το apply θα αποτύγχανε, γιατί το bucket υπάρχει ήδη. Το block `import` του κεφαλαίου 22 φέρνει το υπάρχον bucket στη διεύθυνση που θα είχε αν το είχε φτιάξει το Terraform.

terraform.tfvars

```
sites = {
  docs = { owner = "maria" }
  blog = { owner = "nikos" }
  shop = { owner = "eleni" }
}
```

import.tf

```
import {
  to = module.site["shop"].aws_s3_bucket.this
  id = "tfbook-${data.aws_caller_identity.current.account_id}-shop"
}
```

Το `to` δείχνει ένα resource μέσα σε instance του module, με το κλειδί στο module όπως στο `moved`. Το ίδιο το block όμως γράφεται στο root module. Στο Terraform 1.12 της εξέτασης ένα block `import` μέσα σε child module σταματά με `Import blocks are only allowed in the root module`. Το 1.16 το δέχεται και εκεί, αλλά αυτό είναι εκτός ύλης. Το `id` του bucket είναι το όνομά του. Δεν χρειάζεται να είναι σταθερό κείμενο. Εδώ χτίζεται από το data source, όπως το όνομα στο module.

TERMINAL

```
terraform plan
```

ΕΞΟΔΟΣ

```
…
  # module.site["shop"].aws_s3_bucket.this will be updated in-place
  # (imported from "tfbook-123456789012-shop")
…
      ~ tags                        = {
          + "Owner" = "eleni"
          + "Site"  = "shop"
…
      ~ tags_all                    = {
…
          + "Owner"     = "eleni"
…
          + "Site"      = "shop"
…
  # module.site["shop"].aws_s3_bucket_website_configuration.this will be created
```

```
…
  # module.site["shop"].aws_s3_object.index will be created
…
  # module.site["shop"].aws_ssm_parameter.bucket will be created
…
Plan: 1 to import, 3 to add, 1 to change, 0 to destroy.
…
```

Η περίληψη ξεκινά με ένα τέταρτο μέρος, το `1 to import`. Το bucket εισάγεται και στο ίδιο apply αλλάζει in-place, γιατί το bucket του χεριού δεν έχει tags. Το `tags` αποκτά το `Owner` και το `Site` από το module. Το `tags_all` τα παίρνει κι αυτό, μαζί με τα δύο `default_tags` που η έξοδος εδώ παραλείπει. Τα άλλα τρία resources του site δεν υπάρχουν και δημιουργούνται, `3 to add`.

TERMINAL

```
terraform apply -auto-approve
```

ΕΞΟΔΟΣ

```
…
module.site["shop"].aws_s3_bucket.this: Importing... [id=tfbook-123456789012-shop]
module.site["shop"].aws_s3_bucket.this: Import complete [id=tfbook-123456789012-shop]
module.site["shop"].aws_s3_bucket.this: Modifying... [id=tfbook-123456789012-shop]
module.site["shop"].aws_s3_bucket.this: Modifications complete after 0s [id=tfbook-123456789012-shop]
…
Apply complete! Resources: 1 imported, 3 added, 1 changed, 0 destroyed.
…
```

Το `Importing` και το `Import complete` είναι το import. Το `Modifying` που ακολουθεί είναι το update των tags. Το block `import` έκανε τη δουλειά του και μπορεί να φύγει, όπως είδες στο κεφάλαιο 22.

TERMINAL

```
rm import.tf
terraform plan
```

ΕΞΟΔΟΣ

```
…
No changes. Your infrastructure matches the configuration.
…
```

ΚΡΑΤΑ ΑΥΤΟ

Κάθε αλλαγή σε υπάρχον project περνά από το ίδιο κριτήριο. Διάβασε τις γραμμές με `#` και την περίληψη πριν από το apply. Ένα refactoring θέλει `0 to add` και `0 to destroy`. Ένα import θέλει `to import` και κανένα `destroy`. Ένα drift το βλέπεις πρώτα με `-refresh-only` και μετά διαλέγεις αν θα αλλάξει ο κώδικας ή ο κόσμος.

Καθάρισμα

Στο Moto τίποτα δεν κοστίζει, αλλά το καθάρισμα είναι μέρος της άσκησης. Σε πραγματικό λογαριασμό κάθε ξεχασμένο resource χρεώνεται. Ξεκίνα από το `sites`.

TERMINAL

```
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 12 destroyed.
```

Δώδεκα resources, τέσσερα για καθένα από τα τρία sites. Μαζί τους έφυγε και το bucket του καταστήματος, που δεν το είχε φτιάξει το Terraform. Μετά το import το Terraform το διαχειρίζεται πλήρως.

TERMINAL

```
aws s3api list-objects-v2 --bucket "tfbook-$ACCOUNT-state" --query 'Contents[].Key'
```

ΕΞΟΔΟΣ

```
[
    "sites/terraform.tfstate"
]
```

Το state του `sites` είναι ακόμη στο bucket. Το `destroy` σβήνει τα resources και αφήνει ένα state χωρίς resources. Δεν σβήνει το ίδιο το state. Το S3 δεν σβήνει bucket που έχει αντικείμενα, οπότε σβήσε πρώτα το state με το χέρι. Αυτό επιτρέπεται μόνο τώρα, που το state δεν θυμάται κανένα resource.

TERMINAL

```
aws s3 rm "s3://tfbook-$ACCOUNT-state/sites/terraform.tfstate"
```

ΕΞΟΔΟΣ

```
delete: s3://tfbook-123456789012-state/sites/terraform.tfstate
```

TERMINAL

```
cd ../bootstrap
terraform destroy -auto-approve
```

ΕΞΟΔΟΣ

```
…
Destroy complete! Resources: 1 destroyed.
```

Έλεγξε ότι δεν έμεινε τίποτα στο AWS.

TERMINAL

```
aws s3api list-buckets --query 'Buckets[].Name'
aws ssm describe-parameters --query 'Parameters[].Name'
```

ΕΞΟΔΟΣ

```
[]
[]
```

Καμία λίστα δεν έχει στοιχεία. Στον δίσκο σου έμειναν οι δύο φάκελοι με τον κώδικα, τα lock files και ένα local state του `bootstrap` χωρίς resources. Με αυτά ξαναστήνεις ολόκληρο το project, με τα ίδια βήματα, όποτε το χρειαστείς.

Τι έμαθες

Ο πίνακας συνδέει κάθε κομμάτι του project με το κεφάλαιο που το διδάσκει. Αν ένα βήμα σού φάνηκε δύσκολο, εκεί θα βρεις την εξήγηση.

Βήμα του project	Τι χρησιμοποίησες	Κεφάλαιο
`init`, `plan`, `apply`, `destroy`	Ο κύκλος της υποδομής	1
`fmt -check`, `validate`	Έλεγχος πριν από το plan	2
Ανάγνωση κάθε plan, `-auto-approve`	Σύμβολα, περίληψη, εντολές του plan	3
State χωρίς resources μετά το `destroy`	Το state	4
`required_version`, ακριβής έκδοση, lock file	Providers και εκδόσεις	5
Αναφορές στο `aws_s3_bucket.this.id`	Implicit dependencies	6
`var.sites`, `terraform.tfvars`, `-var`	Input variables	7
`buckets`, `endpoints`, `output -raw`	Outputs	8
`map(object(...))`	Τύποι	9
`for` expressions, `split`, `keys`, `can`	Εκφράσεις και συναρτήσεις	10
`for_each` πάνω σε map, `each.key`	count και for_each	11
`aws_caller_identity`, `aws_ssm_parameters_by_path`	Data sources	12
`validation` με `can(regex(...))`, `check`	Validation και checks	13

Βήμα του project	Τι χρησιμοποίησες	Κεφάλαιο
Module `site`, `for_each` σε module	Modules	15
`default_tags`, buckets, `aws` CLI	Ο AWS provider	17
Bucket του state, `use_lockfile`, `-backend-config`	Backends	19
`plan -refresh-only`, `state list -id`	Drift και εντολές state	21
Block `import` σε instance του module	Import	22
Blocks `moved` με κλειδιά	Refactoring	23
`(sensitive value)` στο parameter	Sensitive τιμές	24

Κάποια κεφάλαια δεν χρειάστηκαν, αλλά έχουν θέση σε ένα πραγματικό project. Το `prevent_destroy` του κεφαλαίου 14 προστατεύει το bucket του state. Για ένα δεύτερο περιβάλλον διαλέγεις ανάμεσα στα workspaces του κεφαλαίου 20 και σε χωριστό key του backend, όπως στην τελευταία άσκηση. Ένα module που θα το μοιραστούν πολλές ομάδες μπαίνει σε Git ή σε registry, όπως στο κεφάλαιο 16. Όταν κάτι αποτύχει χωρίς προφανή λόγο, το `TF_LOG` του κεφαλαίου 25 δείχνει κάθε αίτηση προς το API. Το HCP Terraform του κεφαλαίου 26 τρέχει τα ίδια plans για όλη την ομάδα, με κάθε αλλαγή να ξεκινά από ένα pull request. Το κεφάλαιο 28 έχει τον χάρτη των στόχων της εξέτασης προς τα κεφάλαια.

Ασκήσεις

Κάθε άσκηση είναι επέκταση του project. Ξεκινά από αντίγραφο του φακέλου `sites` όπως έμεινε στο τέλος του κεφαλαίου, χωρίς `.terraform` και χωρίς state, με το lock file. Το bucket του state δεν υπάρχει πια. Επιπλέον το `backend.tf` θα έγραφε στο ίδιο key με το project. Γι' αυτό σβήσε πρώτα το `backend.tf`, εκτός αν η άσκηση λέει άλλο. Το αντίγραφο θα κρατά local state στον δικό του φάκελο. Αν μια άσκηση κάνει apply, κάνε destroy στο τέλος, γιατί όλες φτιάχνουν buckets με τα ίδια ονόματα.

ΑΣΚΗΣΗ 27.1 Κανένα site χωρίς υπεύθυνο

Πρόσθεσε στη μεταβλητή `sites` δεύτερο block `validation` που απορρίπτει `owner` κενό ή μόνο με κενά. Δοκίμασέ το με ένα site `news` που έχει `owner = ""`. Το plan πρέπει να σταματήσει με το δικό σου μήνυμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
╷
│ Error: Invalid value for variable
│
│ Κάθε site θέλει owner.
╵
```

ΑΣΚΗΣΗ 27.2 Τέταρτο site

Πρόσθεσε το site `news` με υπεύθυνο τον `kostas`. Μην κάνεις apply. Τρέξε plan και βρες ποια resources φέρνει η νέα γραμμή του `terraform.tfvars`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # module.site["news"].aws_s3_bucket.this will be created
…
  # module.site["news"].aws_s3_bucket_website_configuration.this will be created
…
  # module.site["news"].aws_s3_object.index will be created
…
  # module.site["news"].aws_ssm_parameter.bucket will be created
…
Plan: 16 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 27.3 Μια ξεχασμένη παράμετρος

Παλιότερα υπήρχε το site `old`. Το bucket του σβήστηκε, αλλά κάποιος ξέχασε το parameter του. Φτιάξε το με το `aws ssm put-parameter`, με όνομα `/sites/old/bucket`, τύπο `String` και όποια τιμή θέλεις. Τρέξε plan και εξήγησε τι βλέπεις. Στο τέλος σβήσε το parameter.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
│ Warning: Check block assertion failed
…
│     │ data.aws_ssm_parameters_by_path.sites.names is list of string with 1 element
…
│ Υπάρχουν parameters στο /sites χωρίς site στη μεταβλητή sites.
…
```

ΑΣΚΗΣΗ 27.4 Το parameter στα outputs

Το pipeline θέλει το όνομα του parameter κάθε site χωρίς να το μαντεύει. Πρόσθεσε στο module output `parameter` και στο root module output `parameters`, ένα map από όνομα site σε όνομα parameter. Κάνε apply, δείξε το output σε JSON και κάνε destroy.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
{"blog":"/sites/blog/bucket","docs":"/sites/docs/bucket","shop":"/sites/shop/bucket"}
```

ΑΣΚΗΣΗ 27.5 Μετονομασία μέσα στο module

Το resource `aws_s3_object.index` του module θα λέγεται `aws_s3_object.home`. Κάνε apply, μετονόμασε το resource και γράψε όσα blocks `moved` χρειάζονται ώστε το plan να μη σβήνει τίποτα. Πόσα blocks θέλουν τα τρία sites; Με Moto τρέξε το πρώτο apply δύο φορές, για τα tags των νέων buckets. Στο τέλος κάνε destroy.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
…
  # module.site["blog"].aws_s3_object.index has moved to module.site["blog"].aws_s3_object.home
…
  # module.site["docs"].aws_s3_object.index has moved to module.site["docs"].aws_s3_object.home
…
  # module.site["shop"].aws_s3_object.index has moved to module.site["shop"].aws_s3_object.home
…
Plan: 0 to add, 0 to change, 0 to destroy.
…
```

ΑΣΚΗΣΗ 27.6 Το project σε δεύτερο state

Αυτή τη φορά κράτα το `backend.tf`. Στήσε το project σε νέο bucket state με όνομα `tfbook-` και μετά τον αριθμό του λογαριασμού και το `-state-v2`. Φτιάξε το bucket με το CLI. Το state πρέπει να γραφτεί στο key `v2/terraform.tfstate`, χωρίς να αλλάξεις το `backend.tf`. Κάνε apply, δείξε ποια keys υπάρχουν στο bucket και καθάρισε τα πάντα, μαζί με το bucket.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
[
    "v2/terraform.tfstate"
]
```

Ερωτήσεις πιστοποίησης

ΕΡΩΤΗΣΗ 27.1

Η configuration `sites` κρατά το state σε bucket που το όνομά του περιέχει τον αριθμό του λογαριασμού. Ποιος τρόπος δίνει το όνομα στο S3 backend χωρίς να γραφτεί σταθερό στον κώδικα;

a. `bucket = "tfbook-${data.aws_caller_identity.current.account_id}-state"` μέσα στο block `backend`
b. `bucket = var.state_bucket` στο block `backend` και η τιμή στο `terraform.tfvars`
c. Partial configuration και η τιμή στο `terraform init -backend-config="bucket=..."`
d. Ένα local με το όνομα και `bucket = local.state_bucket` στο block `backend`

ΕΡΩΤΗΣΗ 27.2

Το `aws_ssm_parameter.site` έχει `for_each` με τα κλειδιά `docs` και `blog`. Το μεταφέρεις στο `module "site"`, που έχει `for_each` με τα ίδια κλειδιά, με όνομα `aws_ssm_parameter.bucket`. Ποιο ζευγάρι `from` και `to` σε block `moved` μεταφέρει σωστά το instance `docs`;

a. `from = aws_ssm_parameter.site` και `to = module.site.aws_ssm_parameter.bucket`
b. `from = aws_ssm_parameter.site["docs"]` και `to = module.site["docs"].aws_ssm_parameter.bucket`
c. `from = aws_ssm_parameter.site["docs"]` και `to = module.site.aws_ssm_parameter.bucket["docs"]`
d. Δεν χρειάζεται `moved`, γιατί τα κλειδιά είναι ίδια και το Terraform ταιριάζει μόνο του τα instances

ΕΡΩΤΗΣΗ 27.3

Κάποιος άλλαξε μια ρύθμιση από την κονσόλα. Τρέχεις `terraform apply -refresh-only` και απαντάς `yes`. Τι συμβαίνει;

a. Το state παίρνει τις πραγματικές τιμές και τίποτα δεν αλλάζει στο AWS.
b. Το Terraform επαναφέρει τη ρύθμιση στην τιμή της configuration.
c. Το Terraform γράφει τη νέα τιμή στα αρχεία `.tf`.
d. Το resource που άλλαξε αντικαθίσταται.

ΕΡΩΤΗΣΗ 27.4

Το site `blog` κλείνει. Θέλεις το Terraform να σταματήσει να διαχειρίζεται το `module.site["blog"]` χωρίς να σβηστεί τίποτα στο AWS. Τα άλλα instances του module μένουν όπως είναι. Τι κάνεις;

a. Block `removed` με `from = module.site["blog"]` και `destroy = false`
b. Σβήνεις το `blog` από το `terraform.tfvars` και κάνεις apply
c. `terraform state rm 'module.site["blog"]'` και μετά σβήνεις το `blog` από τη μεταβλητή
d. `terraform destroy -target='module.site["blog"]'`

ΕΡΩΤΗΣΗ 27.5

Σωστό ή λάθος; Με S3 backend το `terraform destroy` σβήνει και το αντικείμενο του state από το bucket.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 27.6

Το check `parameters` του project βρίσκει ένα ορφανό parameter. Ποια ισχύουν; Επίλεξε δύο.

a. Το plan βγάζει `Warning` και ολοκληρώνεται κανονικά.
b. Το apply σταματά πριν αλλάξει οποιοδήποτε resource.
c. Ο έλεγχος γίνεται σε κάθε plan και σε κάθε apply.
d. Το Terraform σβήνει το ορφανό parameter στο επόμενο apply.



ΚΕΦΑΛΑΙΟ 28

Προσομοίωση εξέτασης

Το τελευταίο κεφάλαιο δεν διδάσκει κάτι νέο. Σου λέει πώς είναι η εξέταση Terraform Associate 004, σε ποιο κεφάλαιο βρίσκεται κάθε στόχος της ύλης και σου δίνει μια πλήρη σειρά 57 ερωτήσεων για να τη γράψεις με χρονόμετρο. Οι απαντήσεις, με εξήγηση για κάθε επιλογή, βρίσκονται στον τόμο των λύσεων.

Η εξέταση με μια ματιά

Η πιστοποίηση λέγεται Terraform Associate και η τρέχουσα έκδοση της εξέτασης είναι η 004. Τα πρακτικά στοιχεία τα δημοσιεύει η HashiCorp στη σελίδα `developer.hashicorp.com/certifications/terraform-associate`. Τον Οκτώβριο του 2026 η σελίδα έγραφε τα παρακάτω.

Στοιχείο	Τιμή
Διάρκεια	Μία ώρα
Τρόπος	Online, με επιτήρηση από απόσταση (online proctored)
Τύποι ερωτήσεων	Σωστό ή λάθος, πολλαπλής επιλογής, πολλαπλής απάντησης
Αριθμός ερωτήσεων	Δεν αναφέρεται
Γλώσσα	Αγγλικά
Έκδοση του Terraform	1.12
Κόστος	70,50 δολάρια ΗΠΑ συν τοπικούς φόρους και χρεώσεις, χωρίς δωρεάν επανάληψη
Ισχύς	Δύο χρόνια

Στοιχείο	Τιμή
Προαπαιτούμενα	Βασική χρήση terminal και κατανόηση αρχιτεκτονικής on premises και cloud

Τα πρακτικά στοιχεία αλλάζουν. Πριν κλείσεις ραντεβού, άνοιξε τη σελίδα της HashiCorp και έλεγξε ξανά τιμή, διάρκεια και κανόνες. Μη βασιστείς σε αυτόν τον πίνακα.

Η εξέταση βασίζεται στο Terraform 1.12, ενώ το βιβλίο τρέχει τα παραδείγματά του με το 1.16.4. Ό,τι προστέθηκε μετά την 1.12 σημειώθηκε ως εκτός ύλης στο κεφάλαιο όπου εμφανίστηκε. Όλες οι απαντήσεις αυτού του κεφαλαίου ισχύουν για το 1.12.

Η ύλη έχει οκτώ ενότητες με 37 στόχους, από το 1a έως το 8d. Ο χάρτης πιο κάτω αντιστοιχίζει κάθε στόχο στα κεφάλαια του βιβλίου.

> **ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ**
> Σε σχέση με την έκδοση 003 η HashiCorp σημειώνει τέσσερα νέα θέματα. Το 4f με το `depends_on` και το `create_before_destroy`, το 4g με τα custom conditions, το 4h με τις ephemeral τιμές και τα write-only arguments και το 8c με την οργάνωση workspaces και projects στο HCP Terraform. Η σειρά αυτού του κεφαλαίου τους δίνει λίγο μεγαλύτερο βάρος.

Οι τρεις τύποι ερωτήσεων

Η HashiCorp δημοσιεύει μαζί με την ύλη δείγματα ερωτήσεων για την 004. Σε αυτά εμφανίζονται τρεις τύποι.

- **Σωστό ή λάθος** (true or false). Μια πρόταση και δύο επιλογές.
- **Πολλαπλής επιλογής** (multiple choice). Μία σωστή απάντηση από μια λίστα.
- **Πολλαπλής απάντησης** (multiple answer). Περισσότερες από μία σωστές. Η ερώτηση λέει πόσες πρέπει να διαλέξεις.

Στο βιβλίο οι επιλογές έχουν γράμματα από το a και μετά. Μια ερώτηση πολλαπλής απάντησης τελειώνει με «Επίλεξε δύο» ή «Επίλεξε τρία». Μια ερώτηση σωστού ή λάθους ξεκινά με «Σωστό ή λάθος;» και έχει την επιλογή a για το σωστό και την b για το λάθος.

Η εξέταση γράφεται στα αγγλικά. Γι’ αυτό το βιβλίο κράτησε στα αγγλικά όρους όπως resource, state, backend, workspace και lock file. Στην οθόνη της εξέτασης θα τους δεις γραμμένους ακριβώς έτσι, μαζί με εντολές και μηνύματα του Terraform που έχεις ήδη δει στα κεφάλαια.

Η σελίδα με τα δείγματα σημειώνει επίσης ότι οι ερωτήσεις δεν έχουν σκοπό να σε ξεγελάσουν. Δεν ψάχνεις κρυφό νόημα. Ψάχνεις την επιλογή που συμφωνεί με όσα είδες να κάνει το Terraform.

Πώς χρησιμοποιείς την προσομοίωση

1. Βρες μια ώρα χωρίς διακοπές. Κλείσε το υπόλοιπο βιβλίο, τις σημειώσεις, τον browser και το terminal. Η προσομοίωση μετρά μόνο αν τη γράψεις χωρίς βοήθεια.
2. Βάλε χρονόμετρο στα 60 λεπτά. Οι 57 ερωτήσεις δίνουν λίγο περισσότερο από ένα λεπτό στην καθεμία.
3. Γράψε σε ένα φύλλο τον αριθμό κάθε ερώτησης και το γράμμα ή τα γράμματα που διαλέγεις. Βάλε αστερίσκο σε όσες μάντεψες.
4. Αν μια ερώτηση σε κρατά πάνω από δύο λεπτά, γράψε την πιο πιθανή απάντηση, βάλε αστερίσκο και προχώρα. Γύρνα σε αυτήν στο τέλος, αν μείνει χρόνος.



5. Όταν λήξει ο χρόνος σταματάς. Ό,τι έμεινε αναπάντητο μετρά λάθος.
6. Βαθμολόγησε με τον τόμο των λύσεων. Μια ερώτηση πολλαπλής απάντησης μετρά σωστή μόνο αν βρήκες όλα τα γράμματα και κανένα παραπάνω.
7. Διάβασε ολόκληρη την εξήγηση κάθε λάθους και κάθε αστερίσκου. Η τελευταία γραμμή της λέει τον στόχο της ύλης και το κεφάλαιο που πρέπει να ξαναδιαβάσεις.

Στο τέλος μέτρα τα σωστά ανά ενότητα. Η σειρά έχει 4 ερωτήσεις από την ενότητα 1, 6 από τη 2, 10 από την 3, 14 από την 4, 6 από την 5, 6 από την 6, 4 από την 7 και 7 από την 8. Η κατανομή ακολουθεί τον αριθμό των στόχων κάθε ενότητας, με λίγο μεγαλύτερο βάρος στις ενότητες 4 και 8, όπου βρίσκονται τα νέα θέματα της 004.

Πριν κλείσεις ραντεβού, στόχευσε σε τέσσερις στις πέντε σωστές σε κάθε ενότητα. Είναι μέτρο του βιβλίου και όχι βαθμός βάσης της HashiCorp. Μια ενότητα κάτω από αυτό σημαίνει ότι τα κεφάλαιά της θέλουν δεύτερο πέρασμα.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Μια λάθος απάντηση σου δείχνει κάτι που δεν ήξερες. Μια σωστή που μάντεψες σου δείχνει το ίδιο. Δούλεψε και τις δύο με τον ίδιο τρόπο. Γύρνα στο κεφάλαιο, ξαναδιάβασε τα πλαίσια ΓΙΑ ΤΗΝ ΕΞΕΤΑΣΗ και ΠΑΓΙΔΑ και λύσε ξανά τις ασκήσεις του. Μην ξαναγράψεις αμέσως την ίδια σειρά. Θα θυμάσαι τα γράμματα και όχι τους λόγους.

Πώς διαβάζεις μια ερώτηση

- Διάβασε την εκφώνηση ως το τέλος πριν κοιτάξεις τις επιλογές. Η φράση που κρίνει την απάντηση είναι συχνά η τελευταία, για παράδειγμα «χωρίς να αλλάξει κανένα resource», «πριν από το apply» ή «ποιος τρόπος δεν συνιστάται».
- Πρόσεξε αν η ερώτηση ζητά έναν τρόπο που δουλεύει ή τον προτεινόμενο. Το `-target`, το `terraform state rm` και το `-refresh=false` λύνουν πολλά προβλήματα, αλλά σπάνια είναι η προτεινόμενη απάντηση.
- Σε ερώτηση πολλαπλής απάντησης μέτρα πόσες ζητά και διάλεξε ακριβώς τόσες.
- Σε ερώτηση με κώδικα διάβασε πρώτα τι ζητά η εκφώνηση και μετά τον κώδικα. Έτσι ξέρεις τι ψάχνεις μέσα του.
- Απόκλεισε πρώτα όσες επιλογές δεν μπορεί να ισχύουν. Μια σημαία ή μια εντολή που δεν συνάντησες σε κανένα κεφάλαιο, όπως `terraform state unlock` ή `-dry-run`, μάλλον δεν υπάρχει.

- Πρόσεξε αν η ερώτηση μιλά για το CLI ή για το HCP Terraform. Η λέξη workspace σημαίνει άλλο πράγμα σε καθένα, όπως εξήγησε το κεφάλαιο 26.

> **ΠΑΓΙΔΑ**
> Μερικές παγίδες επανέρχονται σε πολλές ερωτήσεις. Το `sensitive` κρύβει μια τιμή από την έξοδο, αλλά η τιμή γράφεται στο state. Το plan κάνει refresh, αλλά δεν γράφει state. Το Terraform δεν κάνει rollback όταν ένα apply αποτύχει στη μέση. Το lock file καταγράφει providers και όχι modules. Μια επιλογή που λέει το αντίθετο είναι σχεδόν πάντα λάθος.

Ο χάρτης της ύλης

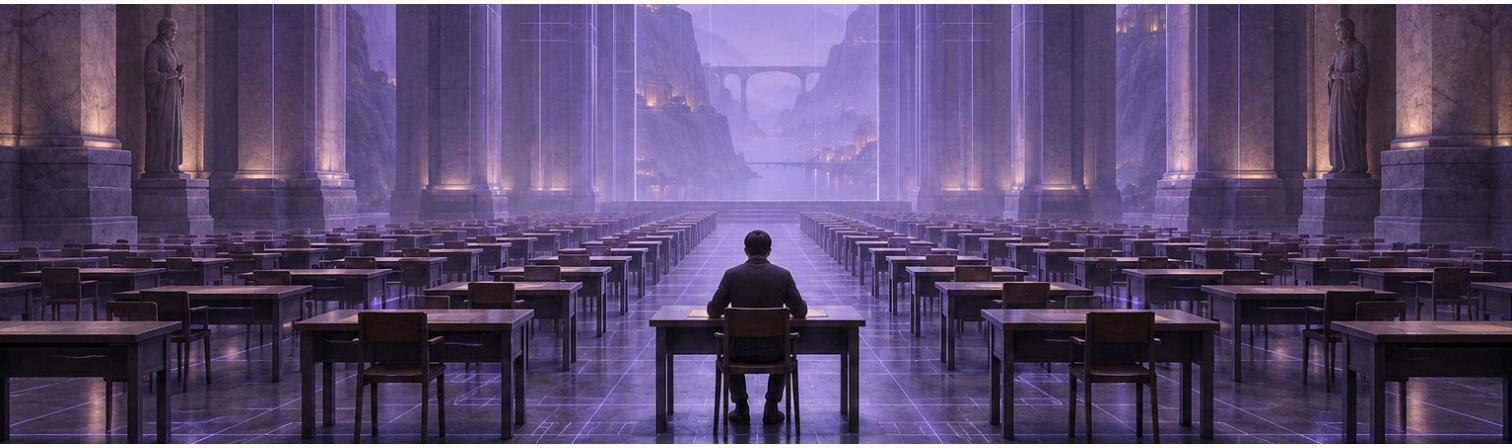
Ο πίνακας δίνει κάθε στόχο της ύλης 004 με μια σύντομη περιγραφή και τα κεφάλαια που τον καλύπτουν. Το κεφάλαιο 27 εμφανίζεται στους στόχους που εξασκεί το τελικό project. Εκεί τους βλέπεις μαζί σε μία configuration.

Στόχος	Κεφάλαια	Τι ζητά
1		**Infrastructure as Code**
1a	1	Τι είναι το Infrastructure as Code
1b	1	Τα πλεονεκτήματα του IaC
1c	5, 17	Multi cloud, hybrid cloud και workflows ανεξάρτητα από την υπηρεσία
2		**Βασικές έννοιες του Terraform**
2a	5, 27	Εγκατάσταση providers και οι εκδόσεις τους
2b	5, 17	Πώς χρησιμοποιεί το Terraform τους providers
2c	5, 17	Configuration με πολλούς providers
2d	4	Πώς χρησιμοποιεί και διαχειρίζεται το state
3		**Το βασικό workflow**
3a	1, 27	Το workflow write, plan, apply
3b	1, 5, 27	Αρχικοποίηση του φακέλου εργασίας με `init`
3c	2, 25, 27	Έλεγχος της configuration με `validate`
3d	1, 3, 27	Δημιουργία και ανάγνωση του plan
3e	1, 3, 27	Εφαρμογή αλλαγών με `apply`
3f	1, 3, 27	Καταστροφή της υποδομής με `destroy`
3g	2, 27	Μορφοποίηση με `fmt`
4		**Configuration**
4a	6, 12, 18, 27	Blocks `resource` και `data`
4b	6, 8, 18, 27	Attributes και αναφορές ανάμεσα σε resources

Στόχος	Κεφάλαια	Τι ζητά
4c	7, 8, 27	Μεταβλητές και outputs
4d	9, 11, 27	Σύνθετοι τύποι
4e	10, 11, 27	Εκφράσεις και συναρτήσεις
4f	6, 14, 18	Εξαρτήσεις, `depends_on` και `create_before_destroy`
4g	13, 27	Validation, preconditions, postconditions και check blocks
4h	24	Ευαίσθητα δεδομένα, ephemeral τιμές, write-only arguments και Vault
5		**Modules**
5a	15, 16	Από πού φορτώνονται τα modules
5b	15	Το scope των μεταβλητών στα modules
5c	15, 18, 27	Χρήση modules στη configuration
5d	16	Εκδόσεις modules
6		**Διαχείριση του state**
6a	4, 19, 20	Το local backend
6b	19, 27	State locking
6c	19, 27	Remote state με block `backend`
6d	21, 23, 27	Drift και διαχείριση του state
7		**Συντήρηση της υποδομής**
7a	22, 27	Import υπάρχουσας υποδομής
7b	4, 21, 23, 27	Εντολές του CLI για το state
7c	25	Verbose logging
8		**HCP Terraform**
8a	26	Υποδομή με το HCP Terraform
8b	26	Συνεργασία και governance
8c	26	Workspaces και projects
8d	26	Σύνδεση του CLI με το HCP Terraform

Ερωτήσεις πιστοποίησης

Βάλε το χρονόμετρο στα 60 λεπτά και ξεκίνα από την πρώτη. Οι απαντήσεις είναι στον τόμο των λύσεων.



ΕΡΩΤΗΣΗ 28.1

Μια ομάδα περιγράφει την υποδομή της με Terraform. Ποια πρόταση περιγράφει το δηλωτικό (declarative) μοντέλο που ακολουθεί;

a. Γράφεις με τη σειρά τις κλήσεις στο API που πρέπει να γίνουν.
b. Γράφεις ένα script που ελέγχει τι υπάρχει πριν φτιάξει κάτι νέο.
c. Γράφεις την κατάσταση που θέλεις και το Terraform υπολογίζει τις ενέργειες που τη φέρνουν.
d. Καταγράφεις τα κλικ στην κονσόλα και το Terraform τα επαναλαμβάνει.

ΕΡΩΤΗΣΗ 28.2

Μια ομάδα ακολουθεί το workflow write, plan, apply. Σε ποιο βήμα βλέπουν οι συνάδελφοι τι ακριβώς θα αλλάξει στην υποδομή, πριν αλλάξει οτιδήποτε;

a. Στο write, όταν το `terraform validate` δείχνει τις αλλαγές.
b. Στο plan, που δείχνει τις ενέργειες χωρίς να τις εκτελέσει.
c. Στο apply, αφού ολοκληρωθούν οι αλλαγές.
d. Στο `terraform init`, που συγκρίνει τη configuration με την υποδομή.

ΕΡΩΤΗΣΗ 28.3

Η configuration σου χρειάζεται το ID ενός VPC που φτιάχνει και διαχειρίζεται άλλη ομάδα με το δικό της Terraform. Δεν θέλεις η δική σου configuration να μπορεί να το αλλάξει ή να το σβήσει. Τι γράφεις;

a. Ένα block `data "aws_vpc"` που βρίσκει το VPC με φίλτρο.
b. Ένα block `resource "aws_vpc"` με το ίδιο CIDR.
c. Ένα block `import` για το VPC και ένα block `resource` για αυτό.
d. Ένα `depends_on` προς το VPC της άλλης ομάδας.

ΕΡΩΤΗΣΗ 28.4

Ένας φάκελος περιέχει μόνο το αρχείο `main.tf` με τη γραμμή `resource "random_pet" "name" {}`. Δεν υπάρχει block `terraform`. Τι κάνει το `terraform init`;

a. Σταματά με σφάλμα, γιατί λείπει το `required_providers`.
b. Ψάχνει ένα εκτελέσιμο με όνομα `random` στο `PATH`.
c. Εγκαθιστά όλους τους providers του namespace `hashicorp`.
d. Βγάζει από το πρόθεμα `random` του τύπου ότι χρειάζεται τον `hashicorp/random` και εγκαθιστά την πιο πρόσφατη έκδοση.

ΕΡΩΤΗΣΗ 28.5

Ποια από τα παρακάτω κάνει το `terraform init`; Επίλεξε τρία.

a. Κατεβάζει τους providers και γράφει τις εκδόσεις τους στο `.terraform.lock.hcl`.
b. Κατεβάζει τα modules που καλεί η configuration.
c. Δημιουργεί τα resources που λείπουν.
d. Ρυθμίζει το backend όπου θα ζει το state.
e. Μορφοποιεί τα αρχεία `.tf` του φακέλου.

ΕΡΩΤΗΣΗ 28.6

Η configuration έχει `resource "random_pet" "server" {}`. Θέλεις το `filename` ενός `local_file` να είναι το όνομα που παρήγαγε ο `random_pet`, με κατάληξη `.txt`. Ποια τιμή γράφεις;

a. `"random_pet.server.id.txt"`
b. `random_pet.server.id + ".txt"`
c. `"${random_pet.server.id}.txt"`
d. `"${random_pet.server}.txt"`

ΕΡΩΤΗΣΗ 28.7

Ένα repository Git έχει πολλά modules σε υποφακέλους. Θέλεις το module του φακέλου `modules/vpc` στο tag `v2.0.0`. Ποιο `source` είναι σωστό;

a. `"git::https://example.com/infra.git/modules/vpc?ref=v2.0.0"`
b. `"git::https://example.com/infra.git//modules/vpc?ref=v2.0.0"`
c. `"git::https://example.com/infra.git?ref=v2.0.0"` μαζί με `path = "modules/vpc"`
d. `"./infra.git/modules/vpc"` μαζί με `version = "2.0.0"`

ΕΡΩΤΗΣΗ 28.8

Δύο μηχανικοί δουλεύουν στην ίδια configuration, ο καθένας από το laptop του. Η configuration δεν έχει block `backend` και ο καθένας κάνει apply από το δικό του αντίγραφο του repository. Ποιο πρόβλημα θα εμφανιστεί;

a. Κάθε laptop έχει δικό του `terraform.tfstate`, οπότε ο ένας δεν βλέπει τι έφτιαξε ο άλλος και τα applies τους συγκρούονται ή φτιάχνουν διπλά resources.
b. Κανένα, γιατί το local backend συγχρονίζει τα states μέσω του provider.
c. Κανένα, γιατί το lock του local backend δουλεύει ανάμεσα σε διαφορετικά μηχανήματα.
d. Το Terraform αρνείται να τρέξει χωρίς block `backend`.

ΕΡΩΤΗΣΗ 28.9

Ποια arguments δέχεται ένα block `variable`; Επίλεξε τρία.

a. `type`
b. `value`
c. `default`
d. `required`
e. `description`

ΕΡΩΤΗΣΗ 28.10

Θέλεις να δεις ποια resources θα σβήσει το `terraform destroy`, χωρίς να σβηστεί τίποτα. Ποια εντολή τρέχεις;

a. `terraform destroy -dry-run`
b. `terraform show -destroy`
c. `terraform state list -destroy`
d. `terraform plan -destroy`

ΕΡΩΤΗΣΗ 28.11

Η configuration έχει δύο ρυθμίσεις του provider `aws`.

```hcl
provider "aws" {
  region = "eu-central-1"
}

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

Πώς φτιάχνεις ένα `aws_s3_bucket` με τη δεύτερη ρύθμιση;

a. Γράφεις `alias = "us"` μέσα στο resource.
b. Γράφεις `provider = aws.us` μέσα στο resource.
c. Γράφεις `provider = us` μέσα στο resource.
d. Γράφεις `providers = { aws = aws.us }` μέσα στο resource.

ΕΡΩΤΗΣΗ 28.12

Θέλεις να φέρεις στο Terraform δύο buckets που φτιάχτηκαν από την κονσόλα.

```hcl
locals {
  buckets = {
    logs   = "acme-logs"
    assets = "acme-assets"
  }
}

import {
  for_each = local.buckets
  to       = aws_s3_bucket.this[each.key]
  id       = each.value
}

resource "aws_s3_bucket" "this" {
  for_each = local.buckets
  bucket   = each.value
}
```

Τι δείχνει το `terraform plan`;

a. Ότι θα γίνει import στα `aws_s3_bucket.this["logs"]` και `aws_s3_bucket.this["assets"]`.
b. Σφάλμα, γιατί ένα block `import` δεν δέχεται `for_each`.
c. Ότι θα δημιουργηθούν δύο νέα buckets.
d. Ότι θα γίνει import σε όλα τα buckets του λογαριασμού.

ΕΡΩΤΗΣΗ 28.13

Δίνεται η μεταβλητή `servers`.

```hcl
variable "servers" {
  type = map(object({
    size = string
    port = number
  }))
  default = {
    web = { size = "small", port = 80 }
    api = { size = "large", port = 8080 }
  }
}
```

Ποια έκφραση δίνει `8080`;

a. `var.servers[1].port`
b. `var.servers["api"].port`
c. `var.servers.port["api"]`
d. `var.servers["api"]["size"]`

ΕΡΩΤΗΣΗ 28.14

Με ποιους τρόπους ξεκινά ένα run σε ένα workspace του HCP Terraform; Επίλεξε τρία.

a. Με commit στο κλαδί ενός repository που είναι συνδεδεμένο με το workspace.
b. Με email προς τη διεύθυνση του workspace.
c. Με `terraform plan` ή `terraform apply` από το CLI, με block `cloud` στη configuration.
d. Με αλλαγή ενός αρχείου στον φάκελο `.terraform` του laptop σου.
e. Με κλήση στο API του HCP Terraform, που ανεβάζει νέο configuration version.

ΕΡΩΤΗΣΗ 28.15

Μια ομάδα κρατά τη configuration του Terraform σε Git και κάθε αλλαγή περνά από pull request. Ποια δύο οφέλη κερδίζει; Επίλεξε δύο.

a. Το Git εφαρμόζει μόνο του κάθε merge στην υποδομή.
b. Κάθε αλλαγή στην υποδομή ελέγχεται από συνάδελφο πριν εφαρμοστεί.
c. Το state δεν χρειάζεται πια, γιατί το Git κρατά το ιστορικό.
d. Το ιστορικό δείχνει ποιος άλλαξε τι στην υποδομή, πότε και γιατί.
e. Τα secrets της configuration προστατεύονται αυτόματα από το Git.

ΕΡΩΤΗΣΗ 28.16

Ένα `terraform apply` πρέπει να φτιάξει τα `local_file.a`, `local_file.b` και `local_file.bad`. Τα δύο πρώτα φτιάχνονται. Το τρίτο αποτυγχάνει με σφάλμα δικαιωμάτων στον δίσκο. Τι ισχύει μετά;

a. Το Terraform σβήνει αυτόματα τα `a` και `b`, ώστε η υποδομή να γυρίσει όπως ήταν.
b. Το state δεν γράφεται, οπότε το επόμενο plan προτείνει ξανά και τα τρία.
c. Τα `a` και `b` υπάρχουν και είναι στο state. Το επόμενο plan προτείνει μόνο το `bad`.
d. Τα `a` και `b` υπάρχουν, αλλά λείπουν από το state μέχρι να τρέξεις `terraform apply -refresh-only`.

ΕΡΩΤΗΣΗ 28.17

Τι επιστρέφει η έκφραση `merge({ a = 1, b = 2 }, { b = 3, c = 4 })`;

a. `{ a = 1, b = 3, c = 4 }`
b. `{ a = 1, b = 2, c = 4 }`
c. Σφάλμα, γιατί το κλειδί `b` υπάρχει και στα δύο.
d. `{ a = 1, b = [2, 3], c = 4 }`

ΕΡΩΤΗΣΗ 28.18

Το root module καλεί δύο modules.

```hcl
module "network" {
  source = "./modules/network"
}

module "app" {
  source    = "./modules/app"
  subnet_id = module.network.subnet_id
}
```

Τι ισχύει για τη σειρά δημιουργίας;

a. Τα δύο modules φτιάχνονται παράλληλα, γιατί δεν υπάρχει `depends_on`.
b. Τα resources του `app` που χρησιμοποιούν το `subnet_id` περιμένουν όσα χρειάζεται το output `subnet_id` του `network`.
c. Το plan αποτυγχάνει, γιατί ένα module δεν μπορεί να διαβάσει output άλλου module.
d. Πρώτα φτιάχνεται ολόκληρο το `app`, γιατί είναι γραμμένο τελευταίο.

ΕΡΩΤΗΣΗ 28.19

Ποιες εντολές παίρνουν lock στο state πριν δουλέψουν, όταν το backend υποστηρίζει locking; Επίλεξε τρία.

a. `terraform plan`
b. `terraform apply`
c. `terraform fmt`
d. `terraform destroy`
e. `terraform validate`

ΕΡΩΤΗΣΗ 28.20

Δίνεται η παρακάτω configuration.

```hcl
resource "random_pet" "bucket" {}

locals {
  bucket_name = "logs-${random_pet.bucket.id}"
}

resource "local_file" "config" {
  filename = "${path.module}/config.txt"
  content  = local.bucket_name
}
```

Το `local_file.config` δεν αναφέρει απευθείας το `random_pet.bucket`. Τι ισχύει;

a. Τα δύο resources φτιάχνονται παράλληλα, γιατί το local κόβει την εξάρτηση.
b. Χρειάζεται `depends_on = [random_pet.bucket]` στο `local_file.config`.
c. Το plan αποτυγχάνει, γιατί ένα local δεν μπορεί να αναφέρει resource.
d. Το Terraform βλέπει την εξάρτηση μέσα από το local και φτιάχνει πρώτα το `random_pet.bucket`.

ΕΡΩΤΗΣΗ 28.21

Ποια από τα παρακάτω πρέπει να μπουν σε version control μαζί με τη configuration; Επίλεξε δύο.

a. Τα αρχεία `.tf`
b. Ο φάκελος `.terraform`
c. Το `.terraform.lock.hcl`
d. Το `terraform.tfstate`
e. Το `terraform.tfstate.backup`

ΕΡΩΤΗΣΗ 28.22

Ένα pipeline CI ελέγχει κάθε pull request. Θέλει να διαβάζει με πρόγραμμα τα σφάλματα της configuration, με αρχείο και γραμμή για το καθένα, ώστε να τα σημειώνει πάνω στον κώδικα. Ποια εντολή τρέχει;

a. `terraform validate -no-color`
b. `terraform fmt -check -diff`
c. `terraform show -json`
d. `terraform validate -json`

ΕΡΩΤΗΣΗ 28.23

Ένα workspace του HCP Terraform έχει execution mode Remote. Στο laptop σου έχεις ορίσει `AWS_ACCESS_KEY_ID` και `AWS_SECRET_ACCESS_KEY` και τρέχεις `terraform plan` στον φάκελο με το block `cloud`. Ο AWS provider αποτυγχάνει, γιατί δεν βρίσκει credentials. Ποια είναι η αιτία και η λύση;

a. Το block `cloud` θέλει τα credentials του AWS στο argument `token`.
b. Το HCP Terraform διαβάζει μόνο το `~/.aws/credentials` του laptop.
c. Το run τρέχει σε μηχάνημα του HCP Terraform, που δεν βλέπει το shell σου. Τα credentials μπαίνουν ως sensitive environment variables στο workspace ή σε variable set.
d. Το plan χρειάζεται πρώτα `terraform login` με τα κλειδιά του AWS.

ΕΡΩΤΗΣΗ 28.24

Θέλεις, αφού δημιουργηθεί ένα `aws_instance` με όνομα `web`, να βεβαιωθείς ότι πήρε δημόσιο DNS όνομα. Αν δεν πήρε, το apply πρέπει να βγάλει error. Τι γράφεις στο `lifecycle` του resource;

a. Ένα `postcondition` με `condition = self.public_dns != ""`
b. Ένα `precondition` με `condition = self.public_dns != ""`
c. Ένα `postcondition` με `condition = aws_instance.web.public_dns != ""`
d. Ένα `validation` με `condition = self.public_dns != ""`

ΕΡΩΤΗΣΗ 28.25

Θέλεις να δεις όλα τα resources που ανήκουν στο module `network`, όπως τα καταγράφει το state. Ποια εντολή τρέχεις;

a. `terraform state show module.network`
b. `terraform state list module.network`
c. `terraform output module.network`
d. `terraform show module.network`

ΕΡΩΤΗΣΗ 28.26

Ένα child module δηλώνει `variable "name"` χωρίς `default`. Το root module το καλεί με ένα block `module` που γράφει μόνο το `source`. Τι γίνεται;

a. Το Terraform ρωτά στο terminal για την τιμή του `name`.
b. Το child παίρνει αυτόματα την τιμή της `var.name` του root, αν υπάρχει.
c. Το `terraform validate` σταματά με `Missing required argument` για το `name`.
d. Η μεταβλητή παίρνει `null` και το plan συνεχίζει.

ΕΡΩΤΗΣΗ 28.27

Στο plan μιας νέας configuration βλέπεις τη γραμμή `+ id = (known after apply)`. Τι σημαίνει;

a. Η τιμή θα γίνει γνωστή μόνο όταν ο provider δημιουργήσει το αντικείμενο στο apply.
b. Η τιμή είναι sensitive και το Terraform την κρύβει.
c. Το resource δεν έχει `id` και το apply θα αποτύχει.
d. Το refresh απέτυχε και το Terraform θα ξαναδοκιμάσει στο apply.

ΕΡΩΤΗΣΗ 28.28

Η μεταβλητή `token` έχει `ephemeral = true`. Τρέχεις `terraform plan -out=tfplan -var token=abc` και μετά `terraform apply tfplan` χωρίς `-var`. Τι γίνεται;

a. Το apply παίρνει την τιμή από το αρχείο του plan.
b. Το apply χρησιμοποιεί κενό string για τη μεταβλητή.
c. Το apply σταματά με `No value for required variable`, γιατί η ephemeral τιμή δεν γράφτηκε στο αρχείο του plan.
d. Το apply ρωτά στο terminal για την τιμή και συνεχίζει.

ΕΡΩΤΗΣΗ 28.29

Ποιες δύο προτάσεις για το block `backend` ισχύουν; Επίλεξε δύο.

a. Οι τιμές του μπορούν να έρθουν από input variables.
b. Μια configuration έχει το πολύ ένα block `backend`.
c. Γράφεται μέσα σε block `provider`.
d. Μετά από αλλαγή στο block χρειάζεται `terraform init` πριν από το επόμενο plan.
e. Κάθε resource μπορεί να έχει δικό του backend.

ΕΡΩΤΗΣΗ 28.30

Μια ομάδα διαχειρίζεται ήδη το AWS με Terraform. Θέλει να διαχειρίζεται με τον ίδιο τρόπο και τα repositories και τις ομάδες της στο GitHub. Τι χρειάζεται;

a. Ένα άλλο εργαλείο, γιατί το Terraform διαχειρίζεται μόνο υποδομή cloud.
b. Το HCP Terraform, γιατί το CLI δουλεύει μόνο με providers της HashiCorp.
c. Έναν provider για το GitHub. Το workflow `init`, `plan`, `apply` μένει το ίδιο.
d. Ένα module που μεταφράζει τα resources του AWS σε resources του GitHub.

ΕΡΩΤΗΣΗ 28.31

Τι επιστρέφει η έκφραση `[for n in [1, 2, 3, 4] : n * 10 if n % 2 == 0]`;

a. `[10, 30]`
b. `[0, 20, 0, 40]`
c. `{ 2 = 20, 4 = 40 }`
d. `[20, 40]`

ΕΡΩΤΗΣΗ 28.32

Ένα `local_file.notes` υπάρχει και είναι στο state. Σβήνεις το block του από τη configuration και τρέχεις `terraform plan`. Τι δείχνει;

a. Ότι θα σβηστεί, με την εξήγηση `because local_file.notes is not in configuration`.
b. Καμία αλλαγή, γιατί το Terraform αγνοεί resources που δεν βρίσκει στη configuration.
c. Ότι θα αφαιρεθεί από το state, χωρίς να σβηστεί το αρχείο.
d. Σφάλμα, γιατί το αρχείο υπάρχει ακόμη στον δίσκο.

ΕΡΩΤΗΣΗ 28.33

Ένα pipeline CI πρέπει να κάνει apply χωρίς να γράψει κάποιος `yes`. Ποιοι δύο τρόποι το πετυχαίνουν; Επίλεξε δύο.

a. `terraform apply -auto-approve`
b. `terraform apply -input=false`
c. `terraform apply -force`
d. `terraform plan -out=tfplan` και μετά `terraform apply tfplan`
e. `terraform validate` και μετά `terraform apply`

ΕΡΩΤΗΣΗ 28.34

Ποιες δύο προτάσεις για τις policies του HCP Terraform ισχύουν; Επίλεξε δύο.

a. Τρέχουν στο laptop του χρήστη μαζί με το `terraform validate`.
b. Γράφονται σε Sentinel ή σε OPA.
c. Ελέγχουν το plan πριν από το apply και μπορούν να σταματήσουν το run.
d. Διορθώνουν αυτόματα τη configuration ώστε να περάσει τον έλεγχο.
e. Ελέγχονται μόνο μετά το apply, πάνω στο νέο state.

ΕΡΩΤΗΣΗ 28.35

Ποια από τα παρακάτω είναι arguments του ίδιου του block `module` και όχι inputs του module που καλείς; Επίλεξε τρία.

a. `source`
b. `version`
c. `backend`
d. `for_each`
e. `provider_version`

ΕΡΩΤΗΣΗ 28.36

Μετά από κάθε apply θέλεις να ελέγχεις ότι η αρχική σελίδα ενός site απαντά. Το DNS καθυστερεί μερικά λεπτά, οπότε ο έλεγχος αποτυγχάνει συχνά αμέσως μετά το πρώτο apply. Δεν θέλεις αυτό να μπλοκάρει το apply. Τι χρησιμοποιείς;

a. Ένα `postcondition` στο resource του server.
b. Ένα `precondition` στο resource της εγγραφής DNS.
c. Ένα block `check` με scoped data source και `assert`.
d. Ένα block `validation` στη μεταβλητή με το όνομα του site.

ΕΡΩΤΗΣΗ 28.37

Το plan δείχνει ότι θα αφαιρέσει από ένα bucket το tag `Owner`, που πρόσθεσε μια συνάδελφος από την κονσόλα. Η ομάδα αποφάσισε ότι το tag είναι σωστό και πρέπει να μείνει. Τι κάνεις ώστε το Terraform να σταματήσει να το αφαιρεί;

a. Τρέχεις `terraform apply -refresh-only`.
b. Προσθέτεις το tag `Owner` στη configuration του bucket.
c. Τρέχεις τα plans με `-refresh=false`.
d. Τρέχεις `terraform state rm` για το bucket και μετά import.

ΕΡΩΤΗΣΗ 28.38

Σωστό ή λάθος; Μετά το apply που έκανε το import, ένα block `import` μπορεί να μείνει στη configuration. Το επόμενο plan δεν προσπαθεί να κάνει ξανά import.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 28.39

Η μεταβλητή `db_password` έχει `sensitive = true`. Ένα output του root module φτιάχνει από αυτήν ένα connection string και δεν έχει `sensitive`. Τι γίνεται;

a. Το output τυπώνει το connection string κανονικά μετά το apply.
b. Το Terraform αφαιρεί αυτόματα τον κωδικό από το string.
c. Το output παίρνει την τιμή `null`.
d. Το `terraform validate` και το plan σταματούν με `Output refers to sensitive values`, μέχρι να βάλεις `sensitive = true` στο output.

ΕΡΩΤΗΣΗ 28.40

Σωστό ή λάθος; Αφού προσθέσεις ένα resource ενός νέου provider σε configuration που έχει ήδη περάσει από `init`, το `terraform plan` δουλεύει χωρίς νέο `terraform init`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 28.41

Κάποιος αλλάζει από την κονσόλα του AWS ένα security group που διαχειρίζεται ένα workspace του HCP Terraform. Ποια λειτουργία το εντοπίζει χωρίς να περιμένει το επόμενο run της ομάδας;

a. Τα run triggers
b. Το drift detection των health assessments
c. Τα variable sets
d. Το private registry

ΕΡΩΤΗΣΗ 28.42

Δίνεται το block `terraform` μιας configuration.

```hcl
terraform {
  required_version = ">= 1.12.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 6.0"
    }
  }
}
```

Ποια πρόταση ισχύει;

a. Το `required_version` ορίζει την έκδοση του AWS provider.
b. Το `~> 6.0` δέχεται μόνο εκδόσεις της σειράς 6.0, όπως η 6.0.3.
c. Το `~> 6.0` δέχεται και την 7.0.0, γιατί το άνω όριο περιλαμβάνεται.
d. Το Terraform CLI πρέπει να είναι 1.12.0 ή νεότερο και ο provider από 6.0 έως πριν από 7.0.

ΕΡΩΤΗΣΗ 28.43

Θέλεις να δεις τις εξαρτήσεις ανάμεσα στα resources μιας configuration σε μορφή που γίνεται διάγραμμα με το Graphviz. Ποια εντολή τρέχεις;

a. `terraform providers`
b. `terraform graph`
c. `terraform show -json`
d. `terraform state list`

ΕΡΩΤΗΣΗ 28.44

Η configuration καλεί ένα module από το Terraform Registry. Πού το αποθηκεύει το `terraform init`;

a. Στο `~/.terraform.d/modules` του χρήστη, κοινό για όλα τα projects.
b. Στον φάκελο `modules` του project, δίπλα στα αρχεία `.tf`.
c. Μέσα στο `terraform.tfstate`.
d. Στο `.terraform/modules` του φακέλου εργασίας.

ΕΡΩΤΗΣΗ 28.45

Σωστό ή λάθος; Το local backend κλειδώνει το state, οπότε δύο εντολές στον ίδιο φάκελο του ίδιου μηχανήματος δεν το αλλάζουν ταυτόχρονα.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 28.46

Ποιες δύο εντολές σβήνουν όλα τα resources που διαχειρίζεται μια configuration; Επίλεξε δύο.

a. `terraform plan -destroy`
b. `terraform destroy`
c. `terraform state rm` για κάθε resource
d. `terraform apply -refresh-only`
e. `terraform apply -destroy`

ΕΡΩΤΗΣΗ 28.47

Ένα `aws_secretsmanager_secret_version` παίρνει τον κωδικό στο write-only argument `secret_string_wo` και έχει `secret_string_wo_version = 1`. Έγινε apply. Ποιες δύο προτάσεις ισχύουν; Επίλεξε δύο.

a. Ο κωδικός δεν βρίσκεται ούτε στο αρχείο του plan ούτε στο state.
b. Αν αλλάξεις μόνο τον κωδικό, το επόμενο plan δείχνει update.
c. Για να σταλεί νέος κωδικός, ανεβάζεις το `secret_string_wo_version` σε `2`.
d. Το plan δείχνει τον κωδικό σε απλό κείμενο, αφού δεν θα αποθηκευτεί.
e. Κάνεις write-only οποιοδήποτε argument αν προσθέσεις `_wo` στο όνομά του.

ΕΡΩΤΗΣΗ 28.48

Ποιες δύο προτάσεις για τα projects του HCP Terraform ισχύουν; Επίλεξε δύο.

a. Κάθε workspace ανήκει σε ακριβώς ένα project.
b. Ένα workspace μπορεί να ανήκει σε πολλά projects ταυτόχρονα.
c. Όλα τα workspaces ενός project μοιράζονται ένα κοινό state.
d. Το δικαίωμα μιας team σε ένα project ισχύει για όλα τα workspaces του, ακόμη και για όσα προστεθούν αργότερα.
e. Το `Default Project` σβήνεται όταν δεν έχει workspaces.

ΕΡΩΤΗΣΗ 28.49

Σωστό ή λάθος; Αφού μια ομάδα περάσει όλη την υποδομή της σε Terraform, οι αλλαγές με το χέρι από την κονσόλα του cloud δεν μπορούν πια να γίνουν.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 28.50

Ορίζεις `TF_LOG=DEBUG` και τρέχεις `terraform plan > plan.txt`. Δεν ορίζεις `TF_LOG_PATH`. Πού βρίσκεις τα logs;

a. Στο `plan.txt`, ανάμεσα στις γραμμές του plan.
b. Στο αρχείο `terraform.log` του φακέλου εργασίας.
c. Στο terminal, γιατί τα logs γράφονται στο stderr.
d. Στο `.terraform/logs`, όπου τα κρατά το init.

ΕΡΩΤΗΣΗ 28.51

Η configuration χρησιμοποιεί τον provider `vault`, που χρειάζεται τη διεύθυνση του server και ένα token. Ποιος τρόπος κρατά το token έξω από τα αρχεία του repository;

a. `token = "hvs.example"` μέσα στο block `provider "vault"`.
b. Άδειο block `provider "vault"` και οι μεταβλητές περιβάλλοντος `VAULT_ADDR` και `VAULT_TOKEN` στο shell ή στο pipeline.
c. Μια γραμμή `vault_token` στο `terraform.tfvars`, που μπαίνει σε Git.
d. Μια μεταβλητή `vault_token` με το token στο `default`.

ΕΡΩΤΗΣΗ 28.52

Το module `hashicorp/dir/template` έχει `version = ">= 1.0.0"`. Το init εγκατέστησε πριν από μήνες την 1.0.1. Σήμερα υπάρχει η 1.0.2. Τι κάνει ένα νέο `terraform init` στον ίδιο φάκελο;

a. Εγκαθιστά την 1.0.2, γιατί το init παίρνει πάντα τη νεότερη επιτρεπτή έκδοση.
b. Αποτυγχάνει, γιατί το lock file κλειδώνει την 1.0.1.
c. Κρατά την 1.0.1. Για την 1.0.2 χρειάζεται `terraform init -upgrade`.
d. Κατεβάζει και τις δύο εκδόσεις και αφήνει το plan να διαλέξει.

ΕΡΩΤΗΣΗ 28.53

Μεταφέρεις το `aws_s3_bucket.logs` του root module μέσα σε ένα νέο module που καλείς με `module "storage"`. Μέσα στο module το resource λέγεται `aws_s3_bucket.this`. Ποιο block, γραμμένο στο root module, αποτρέπει την αντικατάσταση του bucket;

a. `moved` με `from = aws_s3_bucket.logs` και `to = aws_s3_bucket.this`
b. `moved` με `from = module.storage.aws_s3_bucket.this` και `to = aws_s3_bucket.logs`
c. `removed` με `from = aws_s3_bucket.logs`
d. `moved` με `from = aws_s3_bucket.logs` και `to = module.storage.aws_s3_bucket.this`

ΕΡΩΤΗΣΗ 28.54

Σωστό ή λάθος; Το `terraform fmt` μορφοποιεί και τα αρχεία `.tfvars` του φακέλου, όχι μόνο τα `.tf`.

a. Σωστό
b. Λάθος

ΕΡΩΤΗΣΗ 28.55

Το workspace `app-prod` διαβάζει outputs του `network-prod`. Κάθε φορά που αλλάζει το δίκτυο, το `app-prod` πρέπει να ξανατρέχει για να πάρει τις νέες τιμές. Τι ρυθμίζεις;

a. Ένα variable set με τα outputs του `network-prod`.
b. Ένα policy set που ελέγχει το `network-prod`.
c. Ένα run trigger στο `app-prod` με source workspace το `network-prod`.
d. Health assessments στο `app-prod`.

ΕΡΩΤΗΣΗ 28.56

Μια ομάδα σκέφτεται να κρατά το `terraform.tfstate` στο ίδιο repository Git με τον κώδικα. Ποιοι δύο λόγοι το κάνουν κακή ιδέα; Επίλεξε δύο.

a. Το state μπορεί να περιέχει secrets σε απλό κείμενο.
b. Το Git δεν κλειδώνει το state, οπότε δύο συνάδελφοι μπορούν να κάνουν apply από παλιά αντίγραφα.
c. Το Terraform αρνείται να διαβάσει state από φάκελο που είναι repository Git.
d. Το state είναι δυαδικό αρχείο που το Git δεν αποθηκεύει.
e. Το Git σβήνει αυτόματα αρχεία που αλλάζουν σε κάθε commit.

ΕΡΩΤΗΣΗ 28.57

Ένας φάκελος έχει τοπικό `terraform.tfstate` με resources. Προσθέτεις block `cloud` με organization και workspace και θέλεις το state να μεταφερθεί στο workspace. Τι κάνεις;

a. Τρέχεις `terraform init` και απαντάς `yes` όταν ρωτήσει αν θα μεταφέρει το υπάρχον state.
b. Τρέχεις `terraform init -migrate-state`.
c. Τρέχεις `terraform state push terraform.tfstate` πριν από το init.
d. Ανεβάζεις το `terraform.tfstate` στο repository του workspace.

TERRAFORM · ΑΠΟ ΤΟ ΠΡΩΤΟ APPLY ΜΕΧΡΙ ΤΗΝ ΠΙΣΤΟΠΟΙΗΣΗ

Η υποδομή σου σε κώδικα που διαβάζεται, ελέγχεται και ξανατρέχει.

Ένα ελληνικό βιβλίο που ξεκινά από το πρώτο `terraform apply` και φτάνει σε modules, remote state με locking, drift, import, refactoring με `moved`, secrets με Vault και HCP Terraform. Η θεωρία βρίσκεται μέσα στο κεφάλαιο. Κάθε έννοια εμφανίζεται πρώτα σε μικρό παράδειγμα και κάθε εντολή εκτελέστηκε με Terraform 1.16.4, οπότε η έξοδος που διαβάζεις είναι η πραγματική.

Τα πρώτα μέρη δουλεύουν χωρίς cloud. Τα κεφάλαια του AWS τρέχουν σε sandbox λογαριασμό ή τοπικά στο Moto, χωρίς κόστος.

- 28 κεφάλαια σε έξι μέρη
- 172 ασκήσεις με αναμενόμενο αποτέλεσμα
- 223 ερωτήσεις και μια πλήρης προσομοίωση εξέτασης
- όλοι οι στόχοι της εξέτασης Terraform Associate 004
- λύσεις και απαντήσεις με εξηγήσεις σε χωριστό τόμο



ΕΚΔΟΣΗ 1.0 · ΟΚΤΩΒΡΙΟΣ 2026