ITI8801 · Cloud Architectures and DevOps

Infrastructure as Code II: from a spec to modules and shared state

Siim Rebane · TalTech · Autumn 2026

Week 3's configuration, one week later

Seven files, thirty resources, one directory, one state file on one laptop. Then:

- to change the database rule you open the file with the security groups; to see where that subnet id is used you search all seven files
- a second person clones the repository, runs `apply`, and has a state file of their own: two truths about one resource group
- the next assignment needs the same network again, with different machines in it: copy the files and rename every resource

Three problems: **structure** (one flat pile of resources), **shared truth** (whose state is the real one), and **method** (you had a listing to follow; a real task starts from a description of what is needed). By the end of this lecture you can build a system like this from a description, cut it into parts you can reuse, and keep its state where a team can share it.

Building from a spec

A **specification** says what must be true, not how to make it true. For infrastructure it has four parts:

Part	Says
components	what runs: a page, a backend, a database
placement	where each one lives: which network, which subnet
rules	who may talk to whom, and who may not
limits	what the account allows: sizes, counts, cost

An **arrow** is one allowed connection: from the side that opens it to the side that answers, with the port. "Frontend to backend on 8080" is one arrow; a spec is a short list of arrows, and everything not on the list is forbidden.

The order: draw the components and the arrows between them, decide the subnets from the arrows, write the rules that allow the arrows and forbid everything else, then pick sizes that fit the limits.

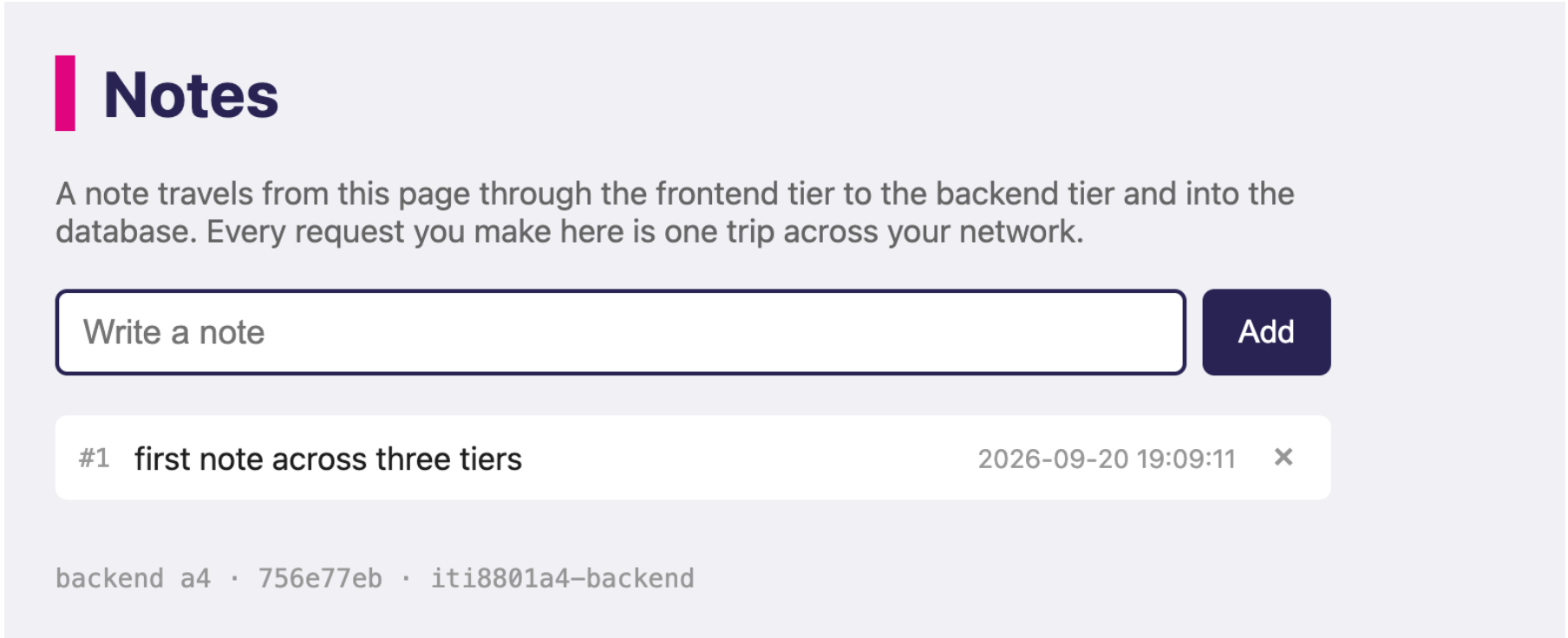
The application: Notes

One image, `ghcr.io/siimrebane/iti8801-notes:a4`, a single Go binary with two modes:

MODE	Does	Listens	Needs
`frontend`	serves the page; forwards `/api/…` to the backend	80	`API_URL`
`api`	stores notes in PostgreSQL	8080	`DATABASE_URL`

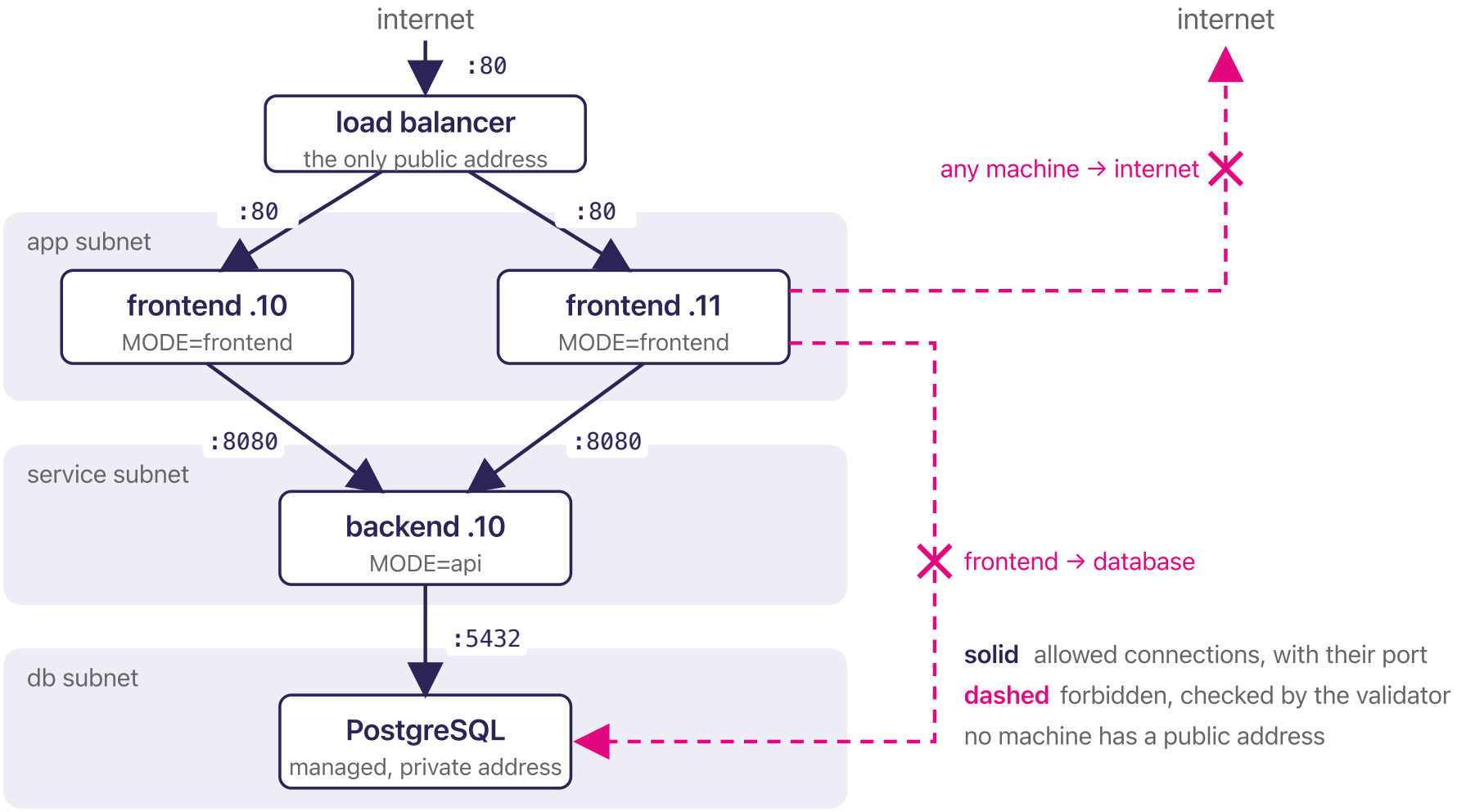
Both modes answer `/health`, `/version`, `/metrics` and `/report`. The report is the self-description the validator reads: which mode, which addresses, what the process can reach and what it cannot. A note typed on the page travels page → frontend → backend → database and back: one request crosses all three tiers.

The page



What the frontend serves. The footer names the backend instance that answered the last request.

The tiers and the arrows



From arrows to subnets

A subnet is the unit a rule can name. Two things that must be treated differently go into different subnets:

Subnet	Holds	In	Out
app	the two frontends	80 from the internet, via the load balancer	the service subnet on 8080
service	the backend	8080 from the app subnet	the db subnet on 5432
db	the managed database	5432 from the service subnet	nothing

The frontends and the backend are both "application VMs", and week 3 would have put them side by side. The spec forbids the frontend from reaching the database, and a rule "5432 from the app subnet" cannot say "but not from those two machines". So the backend gets its own subnet.

The address plan

	Azure `10.60.0.0/16`	AWS `10.61.0.0/16`
app	`10.60.1.0/24`, frontends `.10` `.11`	`10.61.10.0/24`, frontends `.10` `.11`
db	`10.60.2.0/24`	`10.61.20.0/24`, `10.61.21.0/24`
service	`10.60.3.0/24`, backend `.10`	`10.61.30.0/24`, backend `.10`
public	none needed	`10.61.0.0/24`, `10.61.1.0/24`

Two things are fixed, not left to the cloud: the ranges, and the backend's address. The frontends must be told where the backend is before the backend exists, and `10.60.3.10` is a value that can be written into their boot script. A DHCP-assigned address could not.

AWS needs public subnets because an application load balancer and a NAT gateway are placed in subnets; Azure's Standard load balancer is not.

Rules that forbid

Every cloud's default inside a network is permissive: Azure's built-in rule 65000 allows everything from the whole VNet; an AWS security group with no rule denies. "Only from the service subnet" therefore takes two statements on Azure, not one: an allow whose source is that subnet's range, and a deny from `VirtualNetwork` at a lower priority than the built-in allow, so that the built-in never applies. Every "only from" in the spec is one such pair.

On AWS a security group allows what its rules list and nothing else, so there is no deny to write, and a rule may name another security group as its source instead of a range: "from the backend's group" is more precise than "from the service subnet's range", and survives a change of address plan.

Whichever cloud: write the allowed arrows, then check that every other arrow is impossible. The validator tries two of them.

Handing addresses to machines at boot

The frontend needs the backend's address; the backend needs the database's hostname and password. Neither is known when you write the file; both are known to Terraform at apply time. `templatefile` fills them into the boot script:

`templatefile` was lecture 3's tool for this: a boot script with `${…}` holes, filled from Terraform values at plan time, handed to the VM as its cloud-init. The holes this week are the backend's address for the frontends, and the database's hostname, user, password and database name for the backend. The application's README says which environment variable carries each.

The frontend is given `DB_TARGET` too. It is not supposed to reach it, and its report says whether it can.

Limits are part of the spec

Azure for Students allows **6 vCPUs per region**, and every size available to it in this course's regions has 2. Three machines fit. A fourth is refused, whatever it is for:

```
Error: creating Linux Virtual Machine "…-backend":
  OperationNotAllowed: Operation could not be completed as it results in exceeding
  approved Total Regional Cores quota. Current Limit: 6, Current Usage: 6,
  Additional Required: 2, (Minimum) New Limit Required: 8.
```

The spec has three machines: two frontends behind the load balancer and one backend. Read the limits before drawing: `az vm list-usage -l <region>` on Azure, Service Quotas on AWS.

A machine with no way in

Week 3 had a bastion: one more VM whose only job was to be jumped through. This spec has no room for one, and no machine in it is reachable over SSH.

Every cloud VM runs an **agent**, a small program the platform installs, that talks to the platform over an internal address that needs no internet route. The platform can hand it a command:

```
az vm run-command invoke -g <resource group> -n <vm name> \
    --command-id RunShellScript --scripts 'docker logs notes | tail -20'
```

The command runs inside the VM as root and the output comes back through the API. No public IP, no open port, no key on the machine, works even when the subnet has no route out. On AWS the same thing is **Systems Manager**: the agent is on Ubuntu images already, and the instance needs an **instance role** that allows the agent to talk to the service.

Machines get identities too

Lecture 3 defined an identity as who a permission is granted to: a person, or a service principal for Terraform. A VM can be one as well.

On AWS: an **IAM role** whose trust policy lets `ec2.amazonaws.com` assume it, with the managed policy `AmazonSSMManagedInstanceCore` attached, wrapped in an **instance profile** and given to the instance:

```
resource "aws_iam_role" "instance" {
  name = "${var.prefix}-instance-role"
  assume_role_policy = jsonencode({ Version = "2012-10-17", Statement = [{
    Effect = "Allow", Principal = { Service = "ec2.amazonaws.com" }, Action = "sts:AssumeRole" }] })
}
resource "aws_iam_instance_profile" "instance" { name = "${var.prefix}-instance-profile"
                                                 role = aws_iam_role.instance.name }
```

No credentials on the machine: the platform gives the agent short-lived ones because of the role. On Azure the equivalent is a managed identity; run-command needs none, because the agent is trusted by the platform that created the VM.

What the spec looks like when it is met

The only address the internet sees is the load balancer's. When the three tiers are wired, this is what that address answers:

```
$ curl http://<lb address>/version
{"commit":"756e77eb","hostname":"…-frontend-2","mode":"frontend","version":"a4"}

$ curl http://<lb address>/api/version
{"commit":"756e77eb","hostname":"…-backend","mode":"api","version":"a4"}

$ curl -X POST -H 'Content-Type: application/json' \
       -d '{"text":"first note across three tiers"}' http://<lb address>/api/notes
{"id":1,"text":"first note across three tiers","created_at":"2026-09-20T19:09:11.955047Z"}
```

The first answer is a frontend, the second is the backend answering through a frontend's forwarding, the third is a row in the managed database, written by the backend. The validator does the third request and checks the answer.

Modules: the problem

Thirty resources in one directory. Half of them are the network, which stays the same whatever runs in it. Two are the database, which outlives every application that uses it. The rest is the application, which changes every week. A flat directory offers two moves when something must be reused: copy, or edit in place.

What is missing is a way to say "this group of resources is one thing, with these inputs and these results", and to use that thing more than once. In every programming language that is a function. In Terraform it is a **module**.

What a module is

A **module** is a directory of `.tf` files with three kinds of things in it:

Kind	Block	Role
inputs	`variable`	what the caller must or may provide
the work	`resource`, `data`, `locals`	what gets built
outputs	`output`	what the caller may read back

The directory you run `terraform` in is a module too, the **root module**. It becomes a caller by writing a `module` block. Every module has its own variables, its own resources and its own outputs; nothing else is visible across the boundary.

Calling a module

An example, a small site with a web tier and a storage account, nothing to do with this week's spec:

```
module "vnet" {
  source = "./modules/vnet"           # where the directory is

  name                = "site"        # inputs: the module's variables
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
}

module "web" {
  source = "./modules/web"

  subnet_id = module.vnet.web_subnet_id     # an output of the other module
  vm_count  = 2
}
```

`module.vnet.web_subnet_id` is a value that leaves one module and enters another. Terraform sees the reference, so the web module is built after the vnet module's subnet exists. Dependencies cross module boundaries the same way they cross resources: through values.

Inside a module: the inputs

`modules/vnet/variables.tf` of the example:

```
variable "name" { type = string }
variable "resource_group_name" { type = string }
variable "location" { type = string }

variable "address_space" {
  type    = string
  default = "10.10.0.0/16"
}
```

A variable without a default is a required input: the caller must set it or `plan` refuses. A variable with a default is an optional input. The module's variables are its whole interface; a caller cannot reach anything else inside it, and the module cannot see anything of the caller's except what came in this way.

Inside a module: the outputs

`modules/vnet/outputs.tf` of the example:

```
output "web_subnet_id" { value = azurerm_subnet.web.id }
output "address_space" { value = var.address_space }

# A value that must not be handed out before something else exists:
# the ordering travels with the value.
output "private_zone_id" {
  value      = azurerm_private_dns_zone.site.id
  depends_on = [azurerm_private_dns_zone_virtual_network_link.site]
}
```

Outputs are the only way out. Lecture 3 used `depends_on` on a resource that referenced one thing but had to wait for another. Across a module boundary the waiting resource cannot name the thing it waits for, because it is in another module. An output with `depends_on` carries the wait instead: whoever reads `private_zone_id` waits for the link.

Addresses in a configuration with modules

Inside a module, the resources are ordinary resource blocks. What changes is what they may refer to: nothing outside the module except its own variables. What the module should not decide is an input; what others need from it is an output.

Every resource inside a module gets the module's name in front of its address, in the plan, in the state and in every error message:

```
$ terraform state list
azurerm_resource_group.rg
module.vnet.azurerm_virtual_network.site
module.vnet.azurerm_subnet.web
module.web.azurerm_linux_virtual_machine.vm[0]
module.web.azurerm_linux_virtual_machine.vm[1]
```

`module.web.azurerm_linux_virtual_machine.vm[0]` reads: the root module called a module named `web`, and inside it a `count` of VMs, the first one. Every command that takes an address, `state show`, `apply -replace`, `import`, takes this form.

How to split: by lifecycle, not by type

The tempting split is by resource type: all subnets here, all security groups there, all VMs in a third. With that split, changing the application means touching every module.

Split by **what changes together and is destroyed together**. Three lifetimes usually show up in any system:

Lifetime	Changes when
the network	the address plan or the rules change: rarely, and everything else depends on it
the data	the engine or the size changes: almost never, and the data must outlive every application
the application	the image, the machine count, the configuration change: every week

Destroy the application's module and recreate it, and the data keeps its rows and the network keeps its addresses. Where the lines fall in this week's spec is part of the assignment.

Modules from the registry

`source` can also be an address on the public registry:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"

  name = var.prefix
  cidr = "10.61.0.0/16"
  azs  = ["eu-central-1a", "eu-central-1b"]
  private_subnets = ["10.61.10.0/24", "10.61.30.0/24"]
  public_subnets  = ["10.61.0.0/24", "10.61.1.0/24"]
  enable_nat_gateway = true
}
```

Someone else's network module: about a hundred inputs, about two thousand lines, every option AWS has. What it gives: a VPC whose options you did not have to research. What it hides: the two thousand lines themselves. `version` is pinned so that a new release cannot change the plan; the outputs are the only part a caller reads.

What a module must not do

- **Read state it does not own.** A module that looks up "the subnet named app" with a data source instead of taking the id as an input works until two of them exist.
- **Carry credentials.** A password inside a module is a password in every copy of it. It comes in as a `sensitive` variable and goes out nowhere.
- **Decide what the caller should decide.** A hard-coded region, size or name inside a module is a module that only works for its author.
- **Be one resource.** A module around a single `azurerm_subnet` adds an interface and removes nothing.

In a well-cut configuration the root module is the only place that knows the things that differ per person: the name prefix, the subscription, the password, the image. Everything below takes them as inputs.

State, revisited: what week 3 said

The state is Terraform's record of what it built: every resource's real identity and attributes, the mapping between the address in the code and the object in the cloud. Without it, `plan` cannot know that `azurerm_subnet.app` is the subnet with that id, and would create a second one.

Week 3 left the state where `terraform init` puts it: `terraform.tfstate`, next to the code, on the laptop that ran the apply. For one person on one laptop that works.

Two people, one state file

Person A applies from their laptop: 28 resources, state on A's disk. Person B clones the repository, which does not contain the state, and runs `plan`:

```
Plan: 28 to add, 0 to change, 0 to destroy.
```

B's Terraform has an empty state and sees nothing in the cloud that it owns. `apply` fails on the first name that already exists, or, with different names, builds a second copy. If A commits the state file to git to share it, B's apply produces a state that conflicts with A's next commit, and the file also contains the database password in plain text.

The state must live in one place that both read and write, that nobody edits by hand, and that refuses to let two applies run at once.

Remote backends

A **backend** is where Terraform keeps the state and how it locks it. The default, `local`, is the file on disk. The others store it in a service:

Backend	State lives in	Lock
`azurerm`	a blob in a storage account container	a lease on the blob
`s3`	an object in a bucket	a lock file next to the object (Terraform 1.10 and later); before that a DynamoDB table, which still works but is deprecated
`gcs`, `http`, Terraform Cloud, …	their services	their mechanisms

In the root module, on Azure:

```
terraform {
  backend "azurerm" {}     # values come from backend.hcl at init
}
```

The state's home is made before the first `init`

The state's home cannot be in the state. On Azure, once per subscription:

```
az group create -n tfstate-rg -l swedencentral
az storage account create -n tfstate223457 -g tfstate-rg -l swedencentral \
    --sku Standard_LRS --kind StorageV2 --allow-blob-public-access false
az storage container create -n tfstate --account-name tfstate223457 --auth-mode login
```

On AWS, a bucket with versioning on and a DynamoDB table with a `LockID` string key. Once per account, outside the configuration that will use it.

The other way keeps even this in code: a **bootstrap configuration**, a second tiny directory with local state whose only resources are the resource group, the storage account and the container. Its own state file describes nothing but the state's home; if it is lost, the `import` block brings the storage account back under it. Both ways are accepted this week.

Filling `backend.hcl`

Every value in the file names something made in the step above, except the last, which you choose:

Field	Where it comes from
`resource_group_name`	the resource group you created for the state, `tfstate-rg` above
`storage_account_name`	the storage account you created, its globally unique name
`container_name`	the container you created inside it, `tfstate` above
`key`	any name for the blob; one configuration, one key. Two configurations sharing a key overwrite each other's state

```
$ cat backend.hcl
resource_group_name  = "tfstate-rg"
storage_account_name = "tfstate223457"
container_name       = "tfstate"
key                  = "a4.tfstate"
```

The file is not committed: it is per person, and `.gitignore` lists it. On AWS the fields are `bucket`, `key`, `region`, and `dynamodb_table` for the lock table.

`init` with the file

```
$ terraform init -backend-config=backend.hcl
Initializing the backend...
Successfully configured the backend "azurerm"! Terraform will automatically
use this backend unless the backend configuration changes.
```

From here every `plan` and `apply` reads and writes the blob named by `key`, and takes its lease first. Changing any value in the file later means running `init` again; Terraform then asks whether to move the existing state to the new place.

Locking: the collision

Two `plan` commands started one second apart against the same remote state:

```
Error: Error acquiring the state lock

Error message: state blob is already locked
Lock Info:
  ID:        a01fe642-3b41-2f00-9788-4e5e66c9f78c
  Path:      tfstate/a4.tfstate
  Operation: OperationTypePlan
  Who:       student@laptop
  Version:   1.5.7
  Created:   2026-09-20 19:10:41.729575 +0000 UTC
```

The second one refused immediately. Every operation that reads or writes the state takes the lock first and releases it at the end; the lock says who holds it and since when. Two applies cannot interleave.

What a lock protects against, concretely

Without a lock, A and B apply at the same moment:

1. both read the state: 28 resources
2. A creates a VM, writes state: 29 resources
3. B, still working from the copy with 28, creates its VM, writes state: 29 resources, B's

A's VM exists in the cloud and in no state. The next plan wants to create it again; the name exists; apply fails, or a duplicate is made. An orphan that costs money and that nobody's `destroy` will remove.

`-lock-timeout=5m` makes the second command wait instead of refusing. `force-unlock` breaks a lock left behind by a crashed run, and only then: breaking a lock someone holds recreates the situation above.

What is in the state, therefore who may read it

```
$ terraform state pull | grep administrator_password
            "administrator_password": "t2……",      (20 characters, masked here)
```

The database password, in full, in plain text. The private key of every key pair Terraform generated. Every attribute of every resource, sensitive or not, in plain JSON. `sensitive = true` on a variable hides a value in plan output; it does not hide it in the state.

The storage account holding the state therefore holds every secret in the project: no public access, and only the people who may apply may read it. Enable versioning on the container or bucket: every apply writes a new version of the blob, and the previous one is the undo when an apply went wrong.

Moving a state: `init -migrate-state`

A configuration that started with local state, given a backend block later:

```
$ terraform init -backend-config=backend.hcl -migrate-state

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 "azurerm" backend. …
  Enter a value: yes
```

Terraform copies the state file into the blob and stops using the local one. The cloud is untouched; only the record moved. The same command moves between backends. After it, `terraform.tfstate` on disk is a stale copy to delete.

The state as a document you never edit

These commands are the only sanctioned ways to touch it, and every one of them takes the lock:

Command	Does
`terraform state list`	the addresses of everything the state knows
`terraform state show <address>`	one resource's recorded attributes
`terraform state rm <address>`	forget a resource without destroying it
`terraform state mv <from> <to>`	rename in the state, without touching the cloud
`terraform state pull`	print the whole document
`terraform import <address> <id>`	record an existing object in the state, without creating anything

Editing the JSON by hand works until the serial or the lineage no longer matches; then every command refuses. Three of these commands have a form in the code instead: `state mv` is the `moved` block (Terraform 1.1), `terraform import` the `import` block (1.5), `state rm` the `removed` block (1.7).

Drift

Drift is the cloud changing while the code did not: someone opened the portal and clicked. Say the database rule's port was changed from 5432 to 5433 in the portal. The next plan:

```
$ terraform plan
module.network.azurerm_network_security_group.db: Refreshing state...

  # module.network.azurerm_network_security_group.db will be updated in-place
  ~ resource "azurerm_network_security_group" "db" {
      ~ security_rule = [
          - destination_port_range = "5433"
          + destination_port_range = "5432"

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

The plan starts by **refreshing**: reading every known resource from the cloud and comparing. The portal's change shows up as something to undo, because the code says 5432. `apply` put it back in three seconds.

Drift you cannot see

The plan compares the code with what the state knows. A resource created in the portal that the state does not know is invisible:

```
$ az network public-ip create -g <resource group> -n extra-ip --sku Standard
$ terraform plan
No changes. Your infrastructure matches the configuration.
```

The public IP exists, costs money, and no plan will ever mention it. Terraform manages what it created or was told about, nothing else. The reverse is the rule that follows: for what Terraform owns, nobody touches the portal; for what the portal made, either bring it under Terraform or accept that it is invisible.

Import: the situation

The thing exists and should be code. It was clicked together last year; a colleague built it by hand; it was created by another tool. Writing the resource block and running `apply` does not work:

```
azurerm_public_ip.extra: Creating...
Error: a resource with the ID "…/publicIPAddresses/extra-ip" already exists -
to be managed via Terraform this resource needs to be imported into the State.
Please see the resource documentation for "azurerm_public_ip" for more information
```

Terraform wants to create it, because its state has no record. **Import** gives the state that record without creating anything. Two halves: the record in the state, and a block in the code that describes the thing exactly as it is. Import provides the first; you write the second; `plan` tells you whether they match.

Import: the whole sequence

1. **Find the object's id** in the provider's format. Azure: the resource id, from the portal's Properties page or `az <service> show … --query id -o tsv`. AWS: usually the object's own id or name. The provider's documentation page for the resource type ends with the format its import expects.
2. **Write the resource block** in the file and module where the object belongs, describing it as it is, or let `plan -generate-config-out` draft it and cut the draft down.
3. **Add an `import` block** next to it with `to` = the block's address and `id` = the object's id.
4. **`plan`, then `apply`.** The plan says "1 to import, 0 to add"; the apply writes the record into the state. The cloud is untouched.
5. **`plan` again until it says "No changes".** Every difference it shows is between your block and the real object; fix the block, not the object.
6. **Delete the `import` block.** It has done its work; a later reader would otherwise wonder whether the import ever happened. The resource block stays and is from now on an ordinary resource: changed by `apply`, removed by `destroy`.

Import: the block

Terraform 1.5 and later: the import is code.

```
resource "azurerm_public_ip" "extra" {
  name                = "extra-ip"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  allocation_method   = "Static"
  sku                 = "Standard"
}

import {
  to = azurerm_public_ip.extra
  id = "/subscriptions/…/resourceGroups/…/providers/Microsoft.Network/publicIPAddresses/extra-ip"
}
```

`to` is the address in the code, `id` the identity in the cloud, in the provider's format: on Azure the full resource id, on AWS usually the resource's own id or name. The provider's documentation page for every resource ends with the format its import expects.

Import: the plan and the apply

```
$ terraform plan
  # azurerm_public_ip.extra will be imported
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.

$ terraform apply
azurerm_public_ip.extra: Importing... [id=…/publicIPAddresses/extra-ip]
azurerm_public_ip.extra: Import complete
Apply complete! Resources: 1 imported, 0 added, 0 changed, 0 destroyed.

$ terraform plan
No changes. Your infrastructure matches the configuration.
```

The last line is the aim of an import. If the block does not describe the object exactly, the plan after the import shows a change, and that change is the difference between your description and reality. Fix the block until the plan is clean, never apply a change to a freshly imported resource without understanding it: the change is often a destroy and recreate.

Import: letting Terraform write the block

When the object has forty attributes and you know six, Terraform can propose the block. Every attribute comes out, defaults included, and literals where a reference belongs: a starting point, not a result.

```
$ terraform plan -generate-config-out=generated.tf
  # azurerm_public_ip.extra will be imported
  # (config will be generated)
Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.

$ cat generated.tf
# __generated__ by Terraform
# Please review these resources and move them into your main configuration files.
resource "azurerm_public_ip" "extra" {
  allocation_method       = "Static"
  ddos_protection_mode    = "VirtualNetworkInherited"
  idle_timeout_in_minutes = 4
  location                = "swedencentral"
  name                    = "extra-ip"
  resource_group_name     = "…"
  sku                     = "Standard"
  zones                   = []
  …                                      # 9 more, every one a default
}
```

What import cannot do

- **Secrets.** A password the provider cannot read back is imported as unknown; the next plan wants to set it. Set it in the code first.
- **Things the provider does not read.** Some attributes are write-only; the import cannot fill them, and the plan shows a change forever unless the block matches what was written.
- **Objects without a stable identity.** A security rule inside a security group on Azure is part of the group, not its own object; you import the group, with all its rules.
- **A resource inside a module** needs the module address: `to = module.network.azurerm_public_ip.extra`.

And import never changes the cloud. It is a statement about the state only.

`moved`: renaming without destroying

Rename a resource in the code, and the next plan sees one resource gone and one new resource: destroy and create. For a subnet with machines in it that is an outage. The `moved` block tells Terraform it is the same object:

```
resource "azurerm_lb_probe" "version" { … }     # was "http"

moved {
  from = azurerm_lb_probe.http
  to   = azurerm_lb_probe.version
}
```

```
$ terraform plan
  # module.compute.azurerm_lb_probe.http has moved to module.compute.azurerm_lb_probe.version
Plan: 0 to add, 0 to change, 0 to destroy.
```

The state is rewritten, the cloud untouched. The same block moves a resource into a module: `from = azurerm_subnet.app`, `to = module.network.azurerm_subnet.app`. That is how a flat configuration becomes modules with nothing rebuilt.

`removed` : forgetting without destroying

The opposite of import: the object should stay, Terraform should stop managing it. Delete the block and the next plan destroys it. To keep it:

```
removed {
  from = azurerm_public_ip.extra
  lifecycle { destroy = false }
}
```

With the public IP imported above:

```
$ terraform plan
  # azurerm_public_ip.extra will no longer be managed by Terraform, but will not be destroyed
Plan: 0 to add, 0 to change, 0 to destroy.

$ terraform apply
Warning: Some objects will no longer be managed by Terraform
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

The public IP still exists afterwards; Terraform no longer manages it. Terraform 1.7 or later; before that, `terraform state rm <address>`.

When code, state and cloud disagree (1 of 2)

Situation	What happened	`plan` shows	`apply` would	Fix
normal	nothing	`No changes`	nothing	none
drift	someone changed the object in the portal; code and state still say the old value	`1 to change`: the portal's value replaced by the code's	put the object back as the code says	`apply`
unmanaged	an object was made in the portal; neither code nor state knows it	`No changes`: Terraform does not see it	nothing; the object stays, costing money, unseen by any plan	`import` block + a resource block, to manage it; or leave it

The plan compares code with state, then state with cloud. What is in none of the three places Terraform cannot know about.

When code, state and cloud disagree (2 of 2)

Situation	What happened	`plan` shows	`apply` would	Fix
renamed	the resource's name in the code changed; the state records the old name; the object is unchanged	`1 to add, 1 to destroy` : the old name gone, the new one wanted	destroy the object and create a new one	`moved` block: same object, new address; then `plan` shows no changes
given up	the block was deleted from the code; the state still records the object; it must keep existing	`1 to destroy`	destroy the object	`removed` block with `destroy = false` (or `state rm`): the record is dropped, the object stays
deleted by hand	the object was deleted in the portal; code and state still have it	`1 to add` : refresh found it gone	create it again	`apply` , if it should exist; delete the block, if it should stay gone

The rule that avoids most rows: what Terraform owns, only Terraform touches.

Anti-patterns

What people do	What it costs
commit `me.tfvars` with the password in it	the password is in every clone and in the history forever; rotate it
copy a directory instead of writing a module	every fix has to be made twice, and one copy is forgotten
one state for everything: network, database, three applications	every `plan` refreshes everything; every lock blocks everyone; one mistake in one place risks all
`count` for a set of named things	removing the first shifts the others: `[1]` becomes `[0]` and is destroyed and recreated; `for_each` with a map keeps names stable
`apply -auto-approve` in a script, no saved plan	the script applies whatever the plan turned out to be, including a destroy nobody read

Anti-patterns, continued

What people do	What it costs
`apply` from laptops in a team	whose laptop, whose credentials, whose Terraform version; the pipeline applies, people review
`terraform destroy` to fix one resource	`apply -replace=<address>` recreates one; destroy is for the end
editing the state file	a serial that does not match, and every command refuses until the file is restored

Desired state elsewhere

The idea is not Terraform's. Every cloud has its own tool for the same loop: describe, compare, converge.

Tool	Where	Language	State	Reach
Terraform, OpenTofu	any	HCL	a file you keep	every provider with a plugin
CloudFormation	AWS	YAML or JSON	kept by AWS, per stack	AWS
Bicep, ARM	Azure	Bicep, JSON	kept by Azure, per deployment	Azure
Heat	OpenStack	YAML	kept by OpenStack, per stack	OpenStack
Waldur	TalTech's cloud	its own API, a Terraform provider	Waldur	Waldur's services

Why Terraform here, and when not

The cloud's own tool keeps the state for you, so there is no storage account to make and no lock to break, and it knows every feature of its cloud on the day the feature ships. Its limit is the edge of that cloud: a CloudFormation stack cannot create a DNS record at another registrar, a container image in a registry, or anything on Azure.

Terraform keeps the state where you say and reaches everything that has a provider: both clouds in this course, DNS, registries, Kubernetes, GitLab itself, with one language and one workflow. The price is the state and its care.

This course uses Terraform because two clouds are in play and the same files, with different providers, build the same design on both. A team on one cloud with no other systems to describe may reasonably pick the cloud's own tool.

Assignment 4: what to build

The three-tier Notes deployment, from the spec, in your own Terraform:

- at least two modules, split along lines you can defend
- remote state: `backend "azurerm"` or `backend "s3"`, partially configured; the storage made before the first `init`, by hand or by a bootstrap configuration
- the address plan and the rules of the spec
- no public IP on any machine; the load balancer is the only door; port 22 closed
- both tiers running the course image in their modes, the database managed and private

The provider documentation has every resource type. The application's README says what each mode needs.

Assignment 4: what the validator checks

Seven boxes in the order they turn green; six read live, the seventh is your upload.

Box	Proves
Address is in a cloud	the load balancer's IPs are Azure's or AWS's
Frontend tier	two signed frontends answer through the LB, both in the app subnet
Notes round trip	a note posted through the LB is listed and deleted: page → backend → database
Backend tier	the backend's report: SQL works, the database's address is in the db subnet, its own is in the service subnet
Isolation	no frontend reaches the database; no tier reaches the internet
SSH closed	port 22 on the LB address does not answer
Configuration	your `.tf` files: the three tiers' resource types, at least two `module` blocks, a `backend` block; no state, tfvars or key file

Assignment 4: what a red box says

Every box carries the reason the validator saw, in its own words. The first red box is usually the cause of the ones after it; isolation stays red as long as any route to the internet exists.

Box	A reason you may see
Frontend tier	`1 of 2` machines answering; or `addresses: 172.17.0.2`, the container's address instead of the VM's
Notes round trip	`POST /api/notes: expected 201 with the note; got HTTP 502 backend unreachable`
Backend tier	`SQL against the database fails`; `the database address … is not inside …`
Isolation	`frontend-1: reaches the database`; `backend reaches 1.1.1.1:443, 8.8.8.8:53`
Configuration	`no backend "azurerm" or backend "s3" block`; `1 module blocks; the assignment asks for at least 2`

Assignment 4: what to submit

- The validator's code of a PASSED check, in Moodle
- Nothing else: the configuration is in the check, the live proof was taken when it ran
- Then destroy it. The state's storage account can stay; it costs nothing to speak of



Questions