GrowthSchool

TERRAFORM

COMPLETE NOTES

WRITE - PLAN - APPLY

Infrastructure as Code: providers, state, modules and CI/CD

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Terraform in order. Every page has explanation, real HCL, a diagram or example, common mistakes, a student lab and a revision snapshot. We provision infra for a "Shop API" throughout. Provider-specific blocks are clearly labelled.

HOW TO USE THESE NOTES

Use a free-tier or local provider to practice safely. Always read the plan before you apply. Never edit the state file by hand. Use pages 33-37 for fast revision before interviews.

1. Why Infrastructure as Code

GrowthSchool

> **IN SIMPLE TERMS**
> Infrastructure as Code (IaC) means describing your servers and cloud resources in text files instead of clicking in a console, so they are repeatable, reviewable, and version-controlled.

* The Problem With Clicking

- Hand-configured servers drift, are undocumented, and cannot be reproduced reliably.
- "It works because someone clicked something months ago" is not an operations strategy.

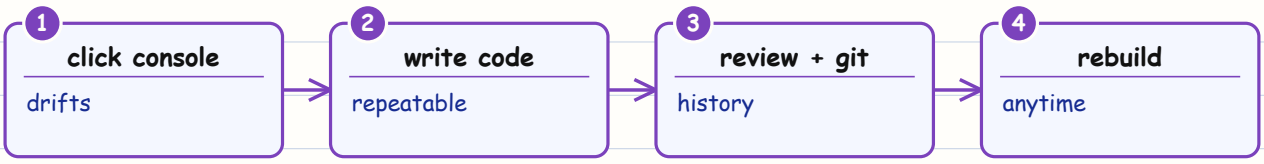
* IaC to the Rescue

- Describe infrastructure in version-controlled code; recreate identical environments on demand.
- You get review, history, repeatability and automation - the same discipline as application code.

* Why Terraform

- Terraform is declarative and cloud-agnostic: one workflow across many providers via plugins.

Why IaC



> **REAL-WORLD EXAMPLE**

```
# infra as code, reviewed and versioned in git
# describe WHAT you want; Terraform figures out HOW
resource "aws_s3_bucket" "assets" {
  bucket = "shop-api-assets"
}
```

> **COMMON MISTAKE**
> Mixing manual console changes with Terraform-managed resources. Terraform then fights your clicks on the next apply. Manage a resource in one place: code, not the console.

> **REVISION SNAPSHOT**
>
> **ASK YOURSELF** What concrete benefits does IaC add over clicking in a console?
>
> **DO THIS** List three problems with manual infra that Terraform solves.

2. How Terraform Works

GrowthSchool

IN SIMPLE TERMS
Terraform reads your desired setup, compares it to what already exists, and then creates or changes only what is needed to match - the same run twice does nothing extra.

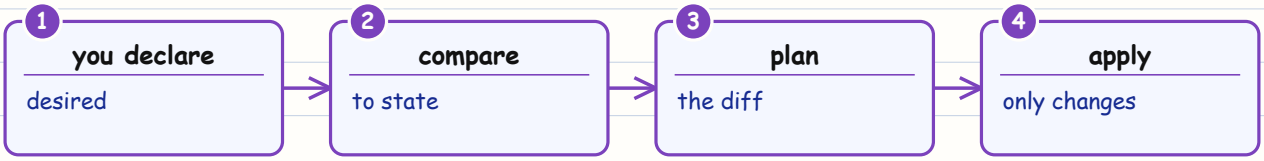
* Declarative & Idempotent

- You declare the desired end state; Terraform computes the changes needed to reach it.
- Running apply repeatedly with no config change makes no changes - it is idempotent.

* Providers & Graph

- Providers are plugins that talk to APIs (AWS, GCP, Azure, Kubernetes, GitHub, and many more).
- Terraform builds a dependency graph and creates/updates resources in the correct order.

How Terraform works



REAL-WORLD EXAMPLE

```
terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 5.0" }
  }
}
provider "aws" { region = "us-east-1" }   # provider-specific
```

REVISION SNAPSHOT

ASK YOURSELF	What does "declarative and idempotent" mean for Terraform?
DO THIS	Explain how Terraform decides the order to create resources.

3. Install, Providers & First Config

GrowthSchool

IN SIMPLE TERMS
Getting started means installing the Terraform tool, declaring which providers (plugins) you need, and running init to download them.

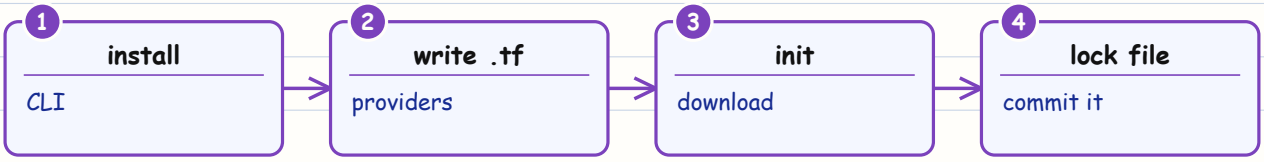
* Get Set Up

- Install the terraform CLI; create a folder with a .tf file; declare required providers.
- terraform init downloads providers into .terraform and writes a lock file.

* The Lock File

- .terraform.lock.hcl pins provider versions for reproducible builds - commit it to Git.

First config



REAL-WORLD EXAMPLE

```
terraform -version
mkdir shop-infra && cd shop-infra
# main.tf declares provider + resources
terraform init
terraform providers
```

COMMON MISTAKE
Not committing .terraform.lock.hcl. Teammates and CI then resolve different provider versions, causing "works on my machine" plan differences. Commit the lock file.

REVISION SNAPSHOT

ASK YOURSELF	What does terraform init do, and what should you commit from it?
DO THIS	Initialise a new config and inspect the generated lock file.

4. Core Workflow

GrowthSchool

IN SIMPLE TERMS
The core workflow is the four commands you repeat: init (set up), plan (preview changes), apply (make them real), and destroy (tear down).

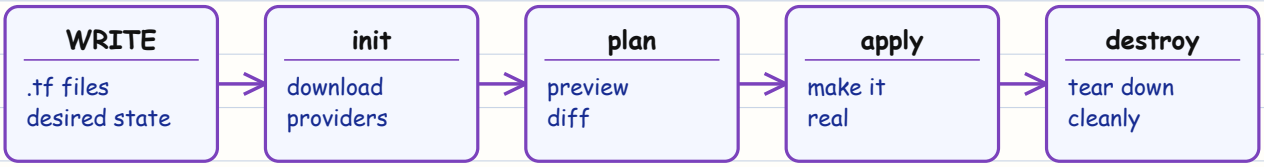
* init -> plan -> apply

- init sets up providers/backend; plan previews changes; apply makes them real; destroy tears down.
- plan is your safety net: always read it before applying, especially in production.

* Reading a Plan

- + create, ~ update in place, -/+ replace (destroy then create), - destroy. Watch for surprise replaces.

Terraform Core Workflow



REAL-WORLD EXAMPLE

```
terraform init
terraform fmt && terraform validate
terraform plan -out=tfplan
terraform apply tfplan
terraform destroy      # careful in shared envs
```

COMMON MISTAKE
Running apply without reading the plan. A small config change can trigger a destructive replace of a database or disk. Always review the +/~/-/+ symbols first.

REVISION SNAPSHOT

ASK YOURSELF	What do +, ~ and -/+ mean in a Terraform plan?
DO THIS	Apply a change with a saved plan file and read every action symbol.

5. HCL Syntax & Blocks

GrowthSchool

IN SIMPLE TERMS
HCL is Terraform's configuration language, built from simple blocks like resource, variable, and output.

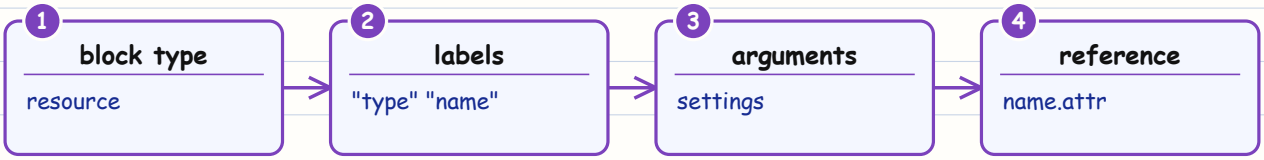
* The Building Blocks

- HCL is built from blocks: block_type "label" "name" { arguments and nested blocks }.
- Resources, variables, outputs, providers, modules and locals are all blocks.

* Reference Style

- Reference values as type.name.attribute, e.g. aws_s3_bucket.assets.arn.
- Comments use # or //; strings support ${...} interpolation.

HCL blocks



REAL-WORLD EXAMPLE

```
resource "aws_instance" "api" {   # block_type "type" "name"
  ami           = var.ami_id
  instance_type = "t3.micro"
  tags = { Name = "shop-api" }
}
# reference: aws_instance.api.public_ip
```

COMMON MISTAKE
Confusing the resource TYPE with its local NAME. In aws_instance.api, aws_instance is the type (fixed by the provider) and api is your chosen name used only for references in this config.

STUDENT LAB
Write one resource block, then in an output reference its attribute using type.name.attribute. Run terraform validate to confirm the reference resolves correctly.

REVISION SNAPSHOT

ASK YOURSELF	What is the structure of an HCL block and a resource reference?
DO THIS	Write a resource block and reference one of its attributes elsewhere.

6. Providers in Depth

GrowthSchool

IN SIMPLE TERMS
A provider is a plugin that lets Terraform talk to a specific platform (AWS, Azure, GCP, Kubernetes, and many more).

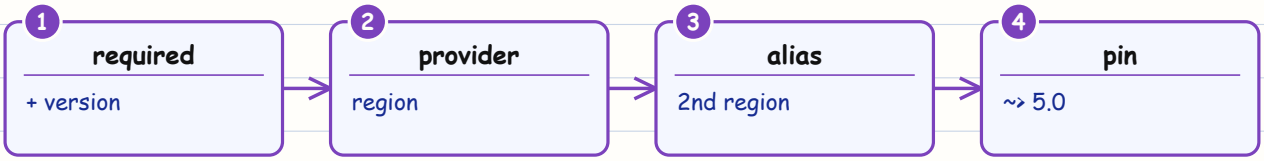
* Configure & Pin

- Declare providers in required_providers and pin versions with constraints like ~> 5.0.
- Configure provider settings (region, credentials source) in a provider block.

* Multiple Providers

- Use alias to configure the same provider more than once, e.g. two regions, and select with provider =.

Providers



REAL-WORLD EXAMPLE

```
provider "aws" {
  region = "us-east-1"
}
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
# resource ... { provider = aws.west }   # provider-specific
```

COMMON MISTAKE
Leaving version constraints open. A new major provider version can change behaviour and break plans. Pin with a sensible constraint (e.g. ~> 5.0) and upgrade deliberately.

REVISION SNAPSHOT

ASK YOURSELF	How do you configure the same provider for two regions?
DO THIS	Pin a provider version and add a second aliased provider.

7. Resources

GrowthSchool

IN SIMPLE TERMS
A resource is a single piece of infrastructure Terraform manages, like a server, bucket, or database.

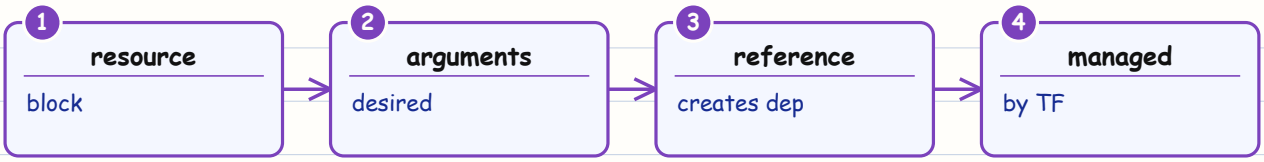
* The Core Object

- A resource block manages one infrastructure object. Terraform creates, updates and deletes it.
- Arguments set desired config; exported attributes (like id, arn) can be referenced by others.

* Implicit Dependencies

- Referencing one resource's attribute in another creates an ordering dependency automatically.
- This implicit graph is why you rarely need explicit depends_on.

A resource



REAL-WORLD EXAMPLE

```
resource "aws_s3_bucket" "assets" {   # provider-specific
  bucket = "shop-api-assets-123"
}
resource "aws_s3_bucket_versioning" "v" {
  bucket = aws_s3_bucket.assets.id   # implicit dependency
  versioning_configuration { status = "Enabled" }
}
```

REVISION SNAPSHOT

ASK YOURSELF	How does Terraform infer the order to create two related resources?
DO THIS	Create two resources where one references the other's attribute.

8. Data Sources

GrowthSchool

IN SIMPLE TERMS
A data source reads information about existing infrastructure without managing it - for example, looking up the latest image ID.

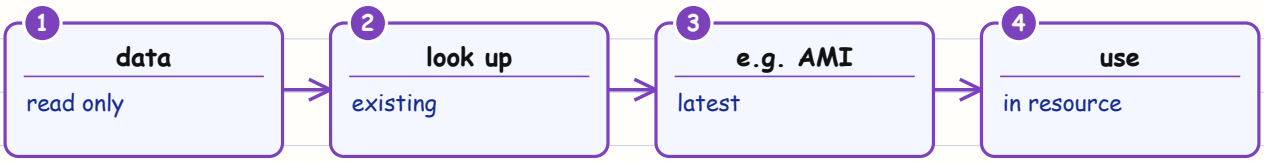
* Read, Do Not Manage

- A data source reads existing infrastructure or provider data without managing its lifecycle.
- Use it to look up an existing VPC, the latest AMI, or your account/region details.

* Why It Matters

- It lets your config depend on things Terraform did not create, keeping values dynamic not hard-coded.

Data source



REAL-WORLD EXAMPLE

```
data "aws_ami" "ubuntu" {          # provider-specific
  most_recent = true
  owners      = ["099720109477"]
  filter { name = "name"
           values = ["ubuntu/*-24.04-amd64-server-*"] }
}
# use: ami = data.aws_ami.ubuntu.id
```

COMMON MISTAKE
Hard-coding an AMI or resource ID that changes over time. Look it up with a data source so the config stays correct as the environment evolves.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a resource and a data source?
DO THIS	Use a data source to fetch a value and feed it into a resource.

9. Input Variables

GrowthSchool

IN SIMPLE TERMS
An input variable is a parameter that makes your configuration reusable, so the same code can build dev, staging, and prod.

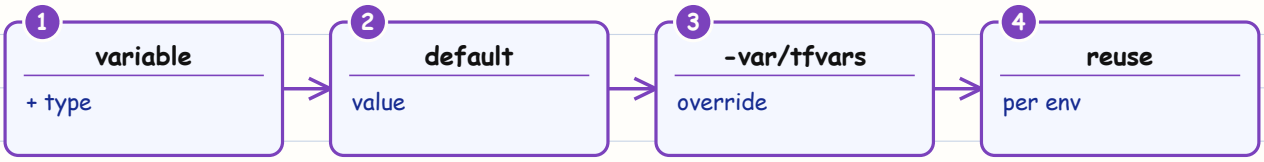
* Parameterise Config

- variable blocks make configs reusable. Set type, description, default and validation.
- Provide values via -var, .tfvars files, or TF_VAR_ environment variables.

* Good Hygiene

- Always type your variables and add descriptions; mark secrets sensitive = true.

Input variables



REAL-WORLD EXAMPLE

```
variable "instance_type" {
  type        = string
  default     = "t3.micro"
  description = "EC2 size for the Shop API"
}
# terraform apply -var="instance_type=t3.small"
# or a prod.tfvars file: instance_type = "t3.large"
```

COMMON MISTAKE
Committing a secrets.tfvars with real credentials. Keep secret tfvars out of Git (.gitignore) and inject them via environment or a secrets manager instead.

REVISION SNAPSHOT

ASK YOURSELF	Name three ways to supply a value for a variable.
DO THIS	Add a typed, described variable and override it with a .tfvars file.

10. Locals

GrowthSchool

IN SIMPLE TERMS
A local is a named value you compute once and reuse throughout a configuration, keeping it tidy (DRY).

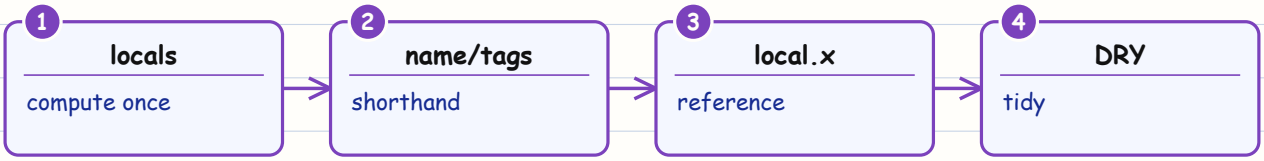
* Named Expressions

- locals define reusable computed values within a module - like constants or shorthands.
- Great for common tags, name prefixes, or expressions used in many places.

* DRY Config

- Change a local once and every reference updates. Reference as local.name.

Locals



REAL-WORLD EXAMPLE

```
locals {
  project = "shop-api"
  common_tags = {
    Project = local.project
    Managed = "terraform"
  }
}
# tags = local.common_tags
```

COMMON MISTAKE
Overusing locals for values that should be input variables. Locals are internal computations; variables are the external interface. Keep the distinction clear.

REVISION SNAPSHOT

ASK YOURSELF	When do you use a local vs an input variable?
DO THIS	Define a common_tags local and apply it to two resources.

11. Outputs

GrowthSchool

IN SIMPLE TERMS
An output exposes a useful value after apply - like a server's IP or URL - and passes values out of a module.

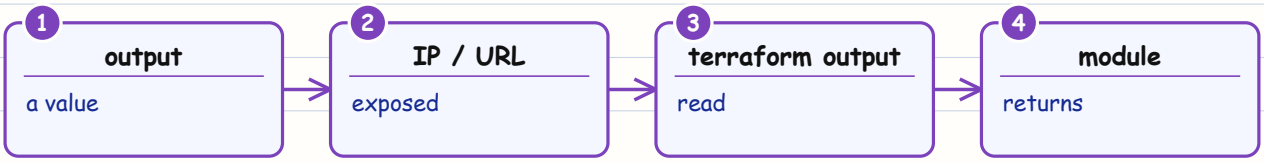
* Expose Results

- output blocks surface useful values after apply: an IP, a URL, an ARN, a database endpoint.
- Outputs are also how a child module returns values to its caller.

* Consume Outputs

- terraform output prints them; other tooling can read them as JSON for downstream steps.

Outputs



REAL-WORLD EXAMPLE

```
output "api_url" {
  value       = "https://${aws_lb.api.dns_name}"
  description = "Public URL of the Shop API"
}
terraform output api_url
terraform output -json | jq .api_url.value
```

COMMON MISTAKE
Printing a secret via an output without sensitive = true. It then appears in plan/apply logs and CI output. Mark secret outputs sensitive so Terraform redacts them.

STUDENT LAB
Add two outputs: one plain (a URL) and one sensitive (a token). Run terraform output and confirm
the sensitive one is redacted unless you request it explicitly.

REVISION SNAPSHOT

ASK YOURSELF	What are the two main uses of outputs?
DO THIS	Expose an API URL as an output and read it with terraform output.

12. State: What & Why

GrowthSchool

IN SIMPLE TERMS
State is the file where Terraform records what it created, so it knows what already exists and what to change next time.

* The Source of Truth

- Terraform records what it manages in a state file mapping config to real resource IDs.
- plan compares config vs state; state lets Terraform know what to update or destroy.

* Handle With Care

- State can contain sensitive values. Never edit it by hand; use terraform state commands.

Terraform reconciles: config vs state vs reality

REAL-WORLD EXAMPLE

```
terraform show           # human-readable state
terraform state list     # managed resources
terraform state show aws_s3_bucket.assets
# NEVER hand-edit terraform.tfstate
```

COMMON MISTAKE
Hand-editing terraform.tfstate to "fix" something. One wrong edit corrupts state and Terraform
may try to destroy/recreate real resources. Use state subcommands instead.

REVISION SNAPSHOT

ASK YOURSELF	Why does Terraform need a state file at all?
DO THIS	List managed resources and inspect one with terraform state show.

13. Remote State & Locking

GrowthSchool

IN SIMPLE TERMS
Remote state stores that file centrally so a whole team can share it safely; locking stops two people applying at the same time.

* Why Remote

- Local state does not work for teams: it is not shared, not backed up, and easy to lose.
- A remote backend stores state centrally, encrypted, and shared across the team and CI.

* Locking

- A lock prevents two people applying at once, which would corrupt state. Most backends lock automatically.

REMOTE BACKEND	STATE LOCKING
state stored centrally (S3, GCS, Azure Blob, Terraform Cloud)	a lock stops two applies at once (DynamoDB, backend-native lock)
shared safely across the team	prevents state corruption
encrypted at rest	force-unlock only when truly stuck

REAL-WORLD EXAMPLE

```
terraform {
  backend "s3" {                  # provider-specific
    bucket         = "shop-tfstate"
    key            = "prod/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "tf-locks"   # state locking
  }
}
```

COMMON MISTAKE
Keeping state on a laptop for a team project. Someone deletes the folder, and now Terraform has no idea what exists. Use a remote backend with locking from day one on shared infra.

REVISION SNAPSHOT

ASK YOURSELF	What two problems does a remote backend with locking solve?
DO THIS	Describe how you would migrate local state to a remote backend.

14. State Safety & Secrets

GrowthSchool

IN SIMPLE TERMS
State safety matters because state can contain secrets in plain text, so you must protect it and keep it out of Git.

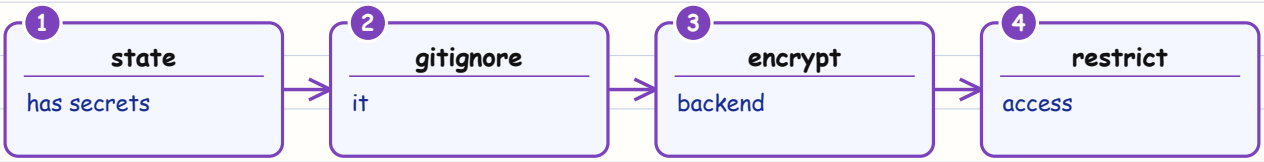
* Secrets Live in State

- Resource attributes (DB passwords, keys) are stored in plaintext inside state.
- Therefore protect state like a secret: encrypt at rest, restrict access, never commit it.

* Safe Practices

- Add terraform.tfstate* to .gitignore; use a backend with encryption + access control + versioning.
- Limit who can read state; prefer generating secrets outside Terraform where practical.

State is sensitive



REAL-WORLD EXAMPLE

```
# .gitignore
terraform.tfstate
terraform.tfstate.backup
.terraform/
*.tfvars   # if they contain secrets
```

COMMON MISTAKE
Committing terraform.tfstate to Git. It often contains secrets in plaintext and is now in history forever. gitignore it, rotate any exposed secret, and move to a secure backend.

REVISION SNAPSHOT

ASK YOURSELF	Why must state be treated as sensitive data?
DO THIS	Confirm your .gitignore excludes state and secret tfvars files.

15. Expressions & Operators

GrowthSchool

IN SIMPLE TERMS
Expressions are how you compute values in HCL using operators, references, and types like lists and maps.

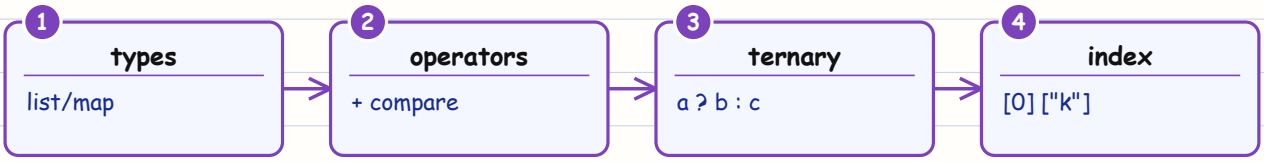
* Work With Values

- HCL supports arithmetic, comparison and logical operators, plus string interpolation "${...}".
- Types include string, number, bool, list, map, set, object and tuple.

* Reference & Index

- Access list items with [0], map values with ["key"], and object attributes with .attr.

Expressions



REAL-WORLD EXAMPLE

```
locals {
  replicas = var.env == "prod" ? 5 : 1
  name     = "${local.project}-${var.env}"
  first_az = var.azs[0]
  port_map = { http = 80, https = 443 }
}
```

COMMON MISTAKE
Mixing types in a comparison or indexing a list that might be empty. Terraform errors at plan time; use length() checks and consistent types to keep expressions robust.

STUDENT LAB
In terraform console, build a map and index it, index a list, and evaluate a ternary. Predict each result before pressing enter to test your mental model of HCL types.

REVISION SNAPSHOT

ASK YOURSELF	Which types and operators does HCL provide?
DO THIS	Write a ternary that sets replicas based on the environment.

16. Built-in Functions

GrowthSchool

IN SIMPLE TERMS
Built-in functions are ready-made helpers (like join, merge, and length) for working with strings and collections.

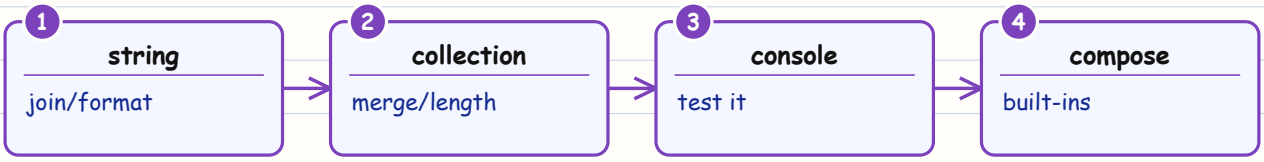
* Common Helpers

- String: upper, lower, format, join, split, replace. Collection: length, merge, concat, lookup, keys.
- There is no user-defined functions - compose the built-ins instead.

* Try It Live

- terraform console is a REPL to test expressions and functions before using them in config.

Functions



REAL-WORLD EXAMPLE

```
terraform console
> merge({a=1}, {b=2})
> join("-", ["shop","api","prod"])
> lookup({http=80,https=443}, "https", 0)
> length(["a","b","c"])
```

COMMON MISTAKE
Building fragile string concatenation when a function like format() or join() is clearer and safer. Explore terraform console to find the right built-in.

REVISION SNAPSHOT

ASK YOURSELF	How do you test an expression without running apply?
DO THIS	Use terraform console to try merge, join and lookup.

17. count

GrowthSchool

IN SIMPLE TERMS
count creates a fixed number of copies of a resource, or switches one on and off with a simple 0-or-1 toggle.

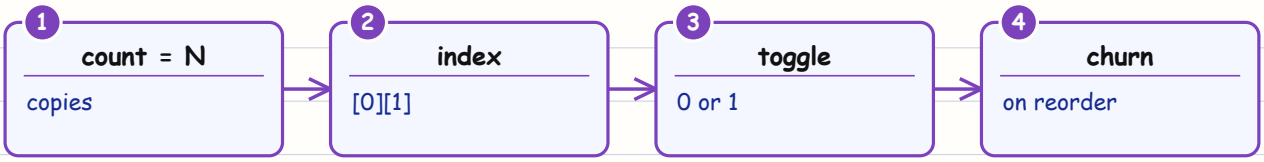
* Repeat a Resource

- count = N creates N instances of a resource, indexed by count.index (0-based).
- Good for identical copies; reference them as name[0], name[1], or name[*] for all.

* Toggle On/Off

- count = var.enabled ? 1 : 0 conditionally creates a resource - a common feature-flag pattern.

count



REAL-WORLD EXAMPLE

```
resource "aws_instance" "worker" {   # provider-specific
  count         = 3
  instance_type = "t3.micro"
  tags = { Name = "worker-${count.index}" }
}
# aws_instance.worker[*].id  -> all ids
```

COMMON MISTAKE
Using count for a set of items that can change order. Removing the middle item re-indexes and churns unrelated resources. Prefer for_each with stable keys for named sets.

REVISION SNAPSHOT

ASK YOURSELF	When does count cause unwanted resource churn?
DO THIS	Create 3 identical resources with count and reference them all.

18. for_each

GrowthSchool

IN SIMPLE TERMS
for_each creates one resource per item in a map or set, giving each a stable key so changes do not disturb the others.

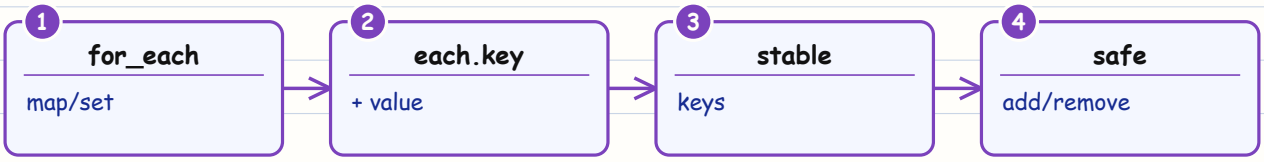
* Create Named Instances

- for_each iterates a map or set, creating one instance per key with each.key / each.value.
- Keys are stable, so adding/removing one item does not disturb the others.

* Prefer Over count

- For a set of distinct, named things (buckets, users, subnets), for_each is safer than count.

for_each



REAL-WORLD EXAMPLE

```
resource "aws_s3_bucket" "b" {        # provider-specific
  for_each = toset(["logs", "assets", "backups"])
  bucket   = "shop-${each.key}"
}
# aws_s3_bucket.b["assets"].arn
```

COMMON MISTAKE
Feeding for_each a value not known until apply (e.g. a generated id). for_each keys must be known at plan time. Use static keys or restructure the data.

REVISION SNAPSHOT

ASK YOURSELF	Why is for_each safer than count for named resources?
DO THIS	Create three buckets from a set and reference one by its key.

19. Conditionals & for Expressions

GrowthSchool

IN SIMPLE TERMS
Conditionals pick a value based on a test; for expressions transform or filter a whole list or map at once.

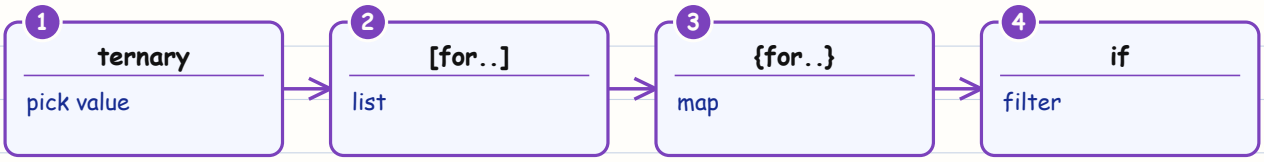
* Conditional Expression

- condition ? true_val : false_val chooses a value inline - great for env-based settings.

* for Expressions

- Transform collections: [for x in list : upper(x)] or {for k,v in map : k => v*2}.
- Filter with an if: [for x in list : x if x != ""].

for expressions



REAL-WORLD EXAMPLE

```
locals {
  tier      = var.env == "prod" ? "large" : "small"
  upper_azs = [for a in var.azs : upper(a)]
  enabled   = { for k,v in var.flags : k => v if v }
}
```

COMMON MISTAKE
Using [ ] for a for expression when you meant to build a map, or { } when you meant a list. Square brackets produce a list/tuple; braces with k => v produce a map/object. Match the shape.

STUDENT LAB
Write one list-comprehension and one map-comprehension over the same input, each with an if filter. Test both in terraform console and confirm the output shapes differ.

REVISION SNAPSHOT

ASK YOURSELF	How do you transform and filter a list in HCL?
DO THIS	Write a for expression that uppercases and filters a list.

20. dynamic Blocks

GrowthSchool

IN SIMPLE TERMS
A dynamic block generates repeated nested settings (like several firewall rules) from a list, instead of writing each by hand.

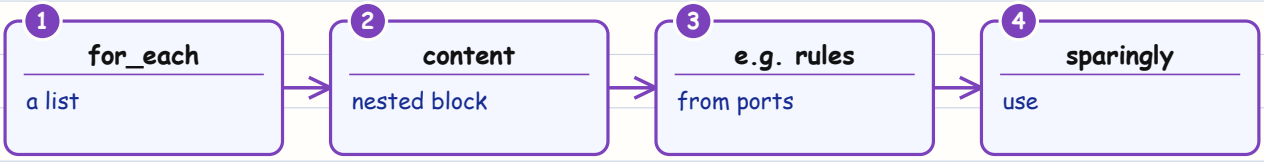
* Generate Nested Blocks

- A dynamic block builds repeated nested blocks (like ingress rules) from a collection.
- Use it when the number of nested blocks depends on a variable.

* Use Sparingly

- Dynamic blocks reduce readability. Reach for them only when static blocks truly cannot express it.

dynamic blocks



REAL-WORLD EXAMPLE

```
resource "aws_security_group" "api" {   # provider-specific
  dynamic "ingress" {
    for_each = var.allowed_ports
    content {
      from_port = ingress.value
      to_port   = ingress.value
      protocol  = "tcp"
    }
  }
}
```

COMMON MISTAKE
Wrapping everything in dynamic blocks until the config is unreadable. If the count is fixed, write the blocks out plainly - clarity beats cleverness.

REVISION SNAPSHOT

ASK YOURSELF	When is a dynamic block the right tool?
DO THIS	Generate security-group rules dynamically from a list of ports.

21. Modules: Basics

GrowthSchool

IN SIMPLE TERMS
A module is a reusable folder of Terraform files you can call with inputs and read outputs from, like a function for infrastructure.

* Reuse Infrastructure

- A module is a folder of .tf files you can call with inputs and read outputs from.
- The top-level config is the root module; it can call child modules to compose infrastructure.

* Call a Module

- Use a module block with source (local path or registry) and pass variables as arguments.

ROOT MODULE	CHILD MODULE
main.tf variables.tf outputs.tf backend.tf calls child modules one per environment	reusable component e.g. modules/network inputs (variables) + outputs versioned + shared

REAL-WORLD EXAMPLE

```
module "network" {
  source     = "./modules/network"
  cidr_block = "10.0.0.0/16"
  env        = var.env
}
# use: module.network.vpc_id
```

REVISION SNAPSHOT

ASK YOURSELF What is the difference between the root module and a child module?

DO THIS Extract a couple of resources into a local module and call it.

22. Module Structure & Registry

GrowthSchool

IN SIMPLE TERMS
A well-structured module has main, variables, and outputs files; the Registry lets you reuse community modules by version.

* Standard Layout

- A module contains main.tf, variables.tf and outputs.tf. Inputs = variables, results = outputs.
- Keep modules focused (one concern) and document their inputs/outputs.

* Sharing

- Pull reusable modules from the Terraform Registry with a versioned source; pin the version.

REAL-WORLD EXAMPLE

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"          # pin the module version
  name    = "shop-vpc"
  cidr    = "10.0.0.0/16"
}
```

COMMON MISTAKE
Using a registry module without pinning version. A new release can change resources and trigger surprise replacements. Always set version and upgrade intentionally.

REVISION SNAPSHOT

ASK YOURSELF	What three files make up a well-structured module?
DO THIS	Consume a pinned registry module and read one of its outputs.

23. Project & Environment Layout

GrowthSchool

IN SIMPLE TERMS
A good project layout keeps each environment (dev, prod) separate with its own state, sharing logic through modules.

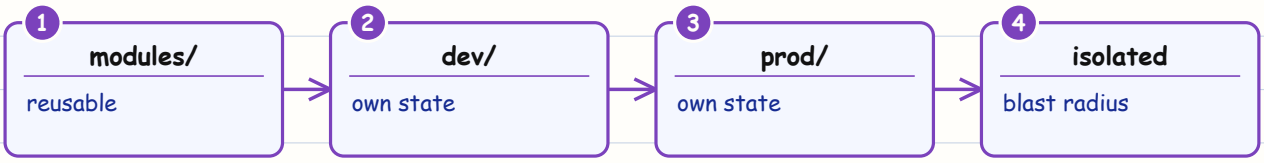
* Separate Environments

- Keep dev, staging and prod isolated with separate state so a mistake in dev cannot harm prod.
- Common pattern: environments/{dev,prod}/ each with its own backend, calling shared modules.

* DRY But Safe

- Share logic through modules; keep environment values in per-env tfvars. Avoid one giant state.

Environment layout



REAL-WORLD EXAMPLE

```
# layout
modules/           # reusable components
environments/dev/  main.tf  dev.tfvars  backend.tf
environments/prod/ main.tf  prod.tfvars backend.tf
# each env: terraform -chdir=environments/prod apply
```

COMMON MISTAKE
Putting all environments in one state. A single apply can then blast every environment at once. Isolate state per environment.

REVISION SNAPSHOT

ASK YOURSELF	Why keep separate state per environment?
DO THIS	Sketch a repo layout with shared modules and per-env configs.

24. Workspaces

GrowthSchool

IN SIMPLE TERMS
A workspace is a way to keep several separate copies of state for the same configuration, for lightweight variants.

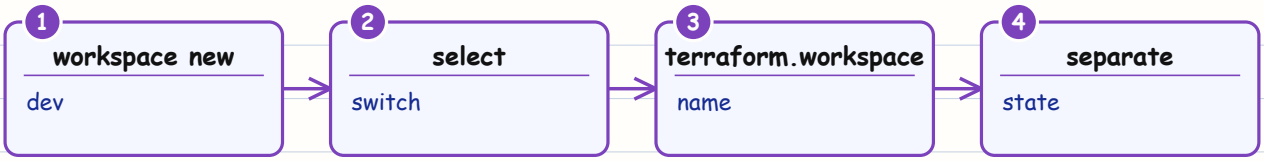
* What They Are

- CLI workspaces keep multiple state instances for the same config (terraform workspace ...).
- Reference the current one via terraform.workspace to vary names/sizes.

* Use With Care

- Workspaces suit lightweight variants; for strongly isolated prod, prefer separate configs/backends.

Workspaces



REAL-WORLD EXAMPLE

```
terraform workspace new dev
terraform workspace select dev
terraform workspace list
# locals { name = "shop-${terraform.workspace}" }
```

COMMON MISTAKE
Using a single set of workspaces as the only separation between dev and prod. They share config and backend; a blast-radius mistake is easy. Prefer separate configs for prod.

REVISION SNAPSHOT

ASK YOURSELF	When are workspaces appropriate vs separate configurations?
DO THIS	Create two workspaces and vary a resource name by workspace.

25. Dependencies & the Graph

GrowthSchool

IN SIMPLE TERMS
Terraform builds a dependency graph from how resources reference each other, so it creates things in the right order automatically.

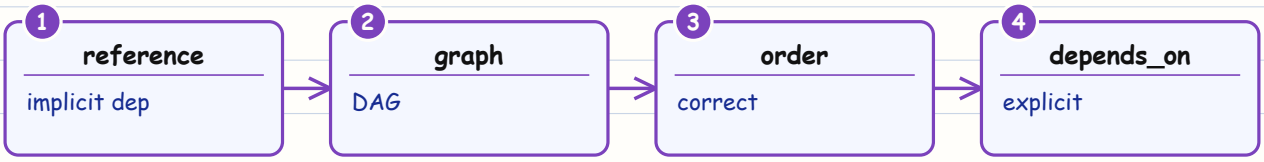
* Implicit First

- Referencing another resource's attribute creates an implicit, correctly-ordered dependency.
- Terraform builds a DAG and parallelises independent work automatically.

* Explicit When Needed

- depends_on forces ordering when there is a hidden dependency Terraform cannot infer from references.

Dependency graph



REAL-WORLD EXAMPLE

```
resource "aws_iam_role_policy_attachment" "a" {  # provider-specific
  # ...
  depends_on = [aws_iam_role.api]   # explicit ordering
}
terraform graph | dot -Tpng > graph.png   # visualise
```

COMMON MISTAKE
Sprinkling depends_on everywhere to "be safe". It hides real relationships and slows applies. Let references create dependencies; use depends_on only for genuine hidden ones.

REVISION SNAPSHOT

ASK YOURSELF	When do you need explicit depends_on?
DO THIS	Generate a dependency graph and identify one implicit edge.

26. Lifecycle Rules

GrowthSchool

IN SIMPLE TERMS
Lifecycle rules change how Terraform handles a resource on updates - for example, creating a replacement before destroying the old one.

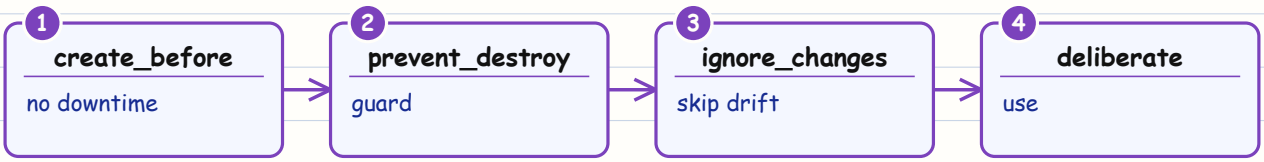
* Control Change Behaviour

- create_before_destroy avoids downtime by making the new resource before removing the old.
- prevent_destroy blocks accidental deletion; ignore_changes ignores drift on chosen attributes.

* Use Deliberately

- These change how apply behaves on sensitive resources like databases and load balancers.

Lifecycle rules



REAL-WORLD EXAMPLE

```
resource "aws_db_instance" "shop" {   # provider-specific
  # ...
  lifecycle {
    prevent_destroy = true
    ignore_changes  = [tags["LastSeen"]]
  }
}
```

COMMON MISTAKE
Leaving prevent_destroy off a production database. One accidental config change that forces replacement can then destroy it. Guard critical stateful resources.

REVISION SNAPSHOT

ASK YOURSELF	What does create_before_destroy prevent, and when use prevent_destroy?
DO THIS	Add prevent_destroy to a critical resource and try to destroy it.

27. Provisioners (Avoid When Possible)

GrowthSchool

IN SIMPLE TERMS
A provisioner runs a script during create or destroy; it is an escape hatch to avoid when a cleaner, declarative option exists.

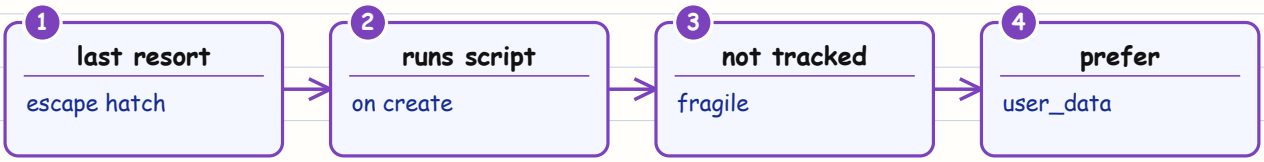
* What They Are

- Provisioners run scripts on create/destroy (local-exec, remote-exec). They are an escape hatch.
- They break the declarative model and are not tracked in state like resources are.

* Prefer Alternatives

- Use cloud-init/user-data, images baked with Packer, or config management instead.
- If you must use one, understand it may not re-run and can leave partial state.

Provisioners



REAL-WORLD EXAMPLE

```
resource "aws_instance" "api" {   # provider-specific
  # prefer user_data over provisioners:
  user_data = file("cloud-init.yaml")
  # last-resort escape hatch:
  # provisioner "local-exec" { command = "echo done" }
}
```

COMMON MISTAKE
Relying on remote-exec provisioners for configuration. They run once, are fragile, and are not reconciled. Bake images or use user_data / config management instead.

REVISION SNAPSHOT

ASK YOURSELF	Why are provisioners a last resort, and what replaces them?
DO THIS	Replace a hypothetical remote-exec setup step with user_data.

28. Import & State Commands

GrowthSchool

IN SIMPLE TERMS
Importing brings an already-existing resource under Terraform's management; state commands let you safely move or remove tracked resources.

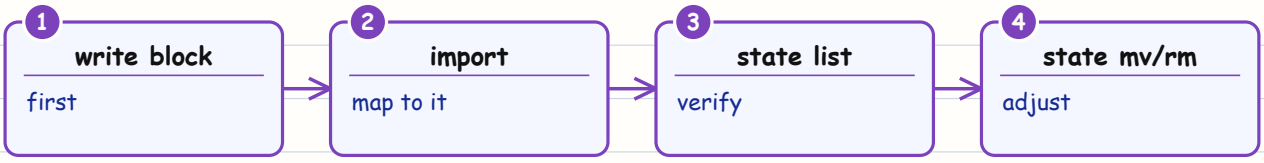
* Adopt Existing Resources

- terraform import brings a manually-created resource under management by mapping it to a config block.
- Newer Terraform also supports import blocks for a plan-reviewed import.

* State Surgery

- state mv renames/moves; state rm stops managing (without deleting); state show inspects.

Import existing



REAL-WORLD EXAMPLE

```
terraform import aws_s3_bucket.assets shop-api-assets-123
terraform state list
terraform state mv aws_s3_bucket.old aws_s3_bucket.assets
terraform state rm aws_s3_bucket.legacy   # stop managing
```

COMMON MISTAKE
Importing without first writing a matching resource block. Import maps to config; with no config, the next plan wants to destroy it. Write the block first, then import.

REVISION SNAPSHOT

ASK YOURSELF	What must exist in config before you terraform import a resource?
DO THIS	Import an existing resource and confirm plan shows no changes.

29. Terraform With a Cloud

GrowthSchool

IN SIMPLE TERMS
Using Terraform with a cloud means the same workflow applies everywhere - only the provider and resource names change per cloud.

* Provider-Specific Reality

- The workflow (init/plan/apply) is identical across clouds; only the provider + resources differ.
- Authenticate via the provider's recommended method (env vars, CLI profile, or CI OIDC role).

* Shop API Example (AWS)

- A minimal stack: an S3 bucket for assets and a security group - everything below is AWS-specific.

PROVIDER-SPECIFIC
Resource names, arguments and auth differ per cloud (AWS/GCP/Azure). The init -> plan -> apply workflow and HCL structure stay the same everywhere.

REAL-WORLD EXAMPLE

```
# ALL PROVIDER-SPECIFIC (AWS)
provider "aws" { region = "us-east-1" }
resource "aws_s3_bucket" "assets" { bucket = "shop-assets-123" }
resource "aws_security_group" "api" {
  name = "shop-api-sg"
  ingress { from_port=443 to_port=443 protocol="tcp"
            cidr_blocks=["0.0.0.0/0"] }
}
```

REVISION SNAPSHOT

ASK YOURSELF	What stays the same and what changes when you switch clouds?
DO THIS	Authenticate to a free-tier cloud and apply one tiny resource.

30. Secrets & Sensitive Values

GrowthSchool

IN SIMPLE TERMS
Sensitive values are secrets you mark so Terraform hides them in output; the real protection is securing the state and using a secrets store.

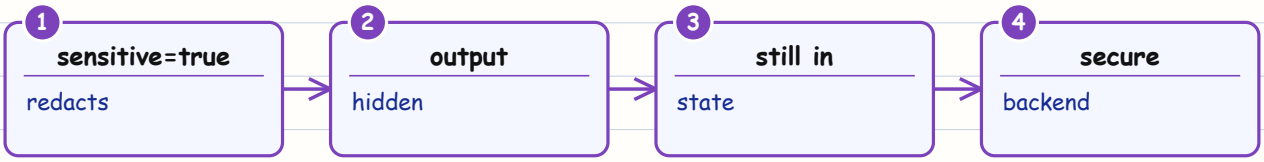
* Mark and Protect

- Set sensitive = true on variables/outputs so Terraform redacts them in plan/apply output.
- Redaction hides them in logs, but they still live in state - protect the backend.

* Source Secrets Well

- Inject via environment (TF_VAR_) or a secrets manager; do not commit secret tfvars.

Sensitive values



REAL-WORLD EXAMPLE

```
variable "db_password" {
  type      = string
  sensitive = true
}
output "db_endpoint" { value = aws_db_instance.shop.address }
# export TF_VAR_db_password=... (from a secrets manager)
```

COMMON MISTAKE
Assuming sensitive = true encrypts the value. It only hides it in CLI output; the value is still plaintext in state. Secure the backend and restrict access.

REVISION SNAPSHOT

ASK YOURSELF	What does sensitive = true actually do (and not do)?
DO THIS	Mark a variable sensitive and confirm it is redacted in plan output.

31. CI/CD & Plan Review

GrowthSchool

IN SIMPLE TERMS
A Terraform pipeline runs checks and a plan on each pull request for review, then applies the change automatically when it is merged.

* Automate Safely

- On a PR: run fmt + validate, then terraform plan and post the diff for human review.
- On merge to main: run apply (often with a required approval) using a saved plan.

* Guardrails

- Use remote state + locking, least-privilege CI credentials (OIDC), and policy checks (OPA/Sentinel).

Terraform CI/CD: plan on PR, apply on merge

REAL-WORLD EXAMPLE

```
# CI steps
terraform fmt -check
terraform validate
terraform plan -out=tfplan   # post summary to the PR
# on merge:
terraform apply tfplan
```

COMMON MISTAKE
Letting CI auto-apply on every push with no plan review. Infra changes deserve the same review as code. Plan on PR, apply on merge with approval.

REVISION SNAPSHOT

ASK YOURSELF	What runs on a PR vs on merge in a Terraform pipeline?
DO THIS	Describe a pipeline that plans on PR and applies on merge.

32. Drift Detection

GrowthSchool

IN SIMPLE TERMS
Drift is when the real infrastructure no longer matches your code (usually from manual changes); plan detects it so you can reconcile.

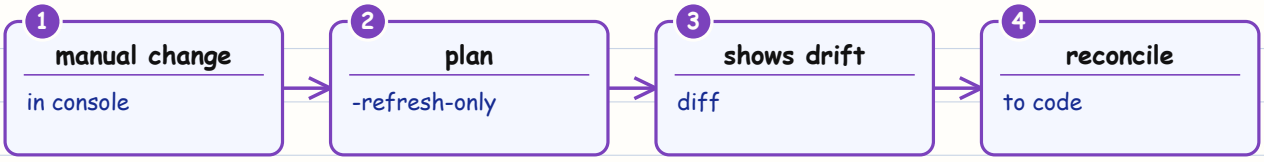
* What Is Drift

- Drift is when real infrastructure differs from state - usually from manual console changes.
- terraform plan detects drift by refreshing and comparing state to reality.

* Respond to Drift

- Decide: reconcile by applying config, or import/adjust config to match an intentional change.
- Prevent drift by managing resources only through Terraform.

Drift detection



REAL-WORLD EXAMPLE

```
terraform plan -refresh-only    # show drift without changes
terraform apply -refresh-only   # update state to match reality
terraform plan                  # then reconcile to config
```

COMMON MISTAKE
Ignoring drift until an apply suddenly wants to undo someone's emergency manual fix. Detect drift regularly and bring changes back into code promptly.

REVISION SNAPSHOT

ASK YOURSELF	What causes drift and how does plan reveal it?
DO THIS	Manually tag a resource in the console, then detect the drift with plan.

33. Troubleshooting

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm way to fix common Terraform issues like a stuck state lock or a value not known until apply.

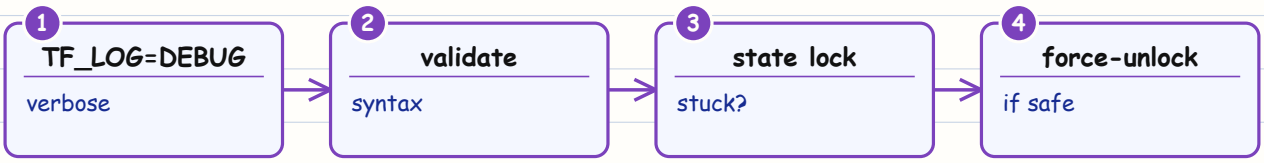
* Common Issues

- State lock stuck: a crashed apply left a lock; investigate, then force-unlock only if truly safe.
- for_each/count value not known at plan time: restructure to use plan-time-known keys.

* Debugging

- Set TF_LOG=DEBUG for verbose provider logs; use terraform validate and console to isolate issues.
- Read the error carefully - Terraform errors usually name the exact resource and attribute.

Troubleshoot



REAL-WORLD EXAMPLE

```
TF_LOG=DEBUG terraform apply
terraform force-unlock <LOCK_ID>   # only when safe
terraform validate
terraform state list | grep <name>
```

COMMON MISTAKE
Running force-unlock reflexively. If another apply is genuinely running, unlocking corrupts state. Confirm no apply is in progress first.

REVISION SNAPSHOT

ASK YOURSELF	When is terraform force-unlock safe to run?
DO THIS	Enable TF_LOG=DEBUG and read where a failing apply spends its time.

34. Best Practices

GrowthSchool

IN SIMPLE TERMS
Best practices are the proven habits - pinned versions, remote locked state, small blast radius, reviewed plans - that keep infra safe.

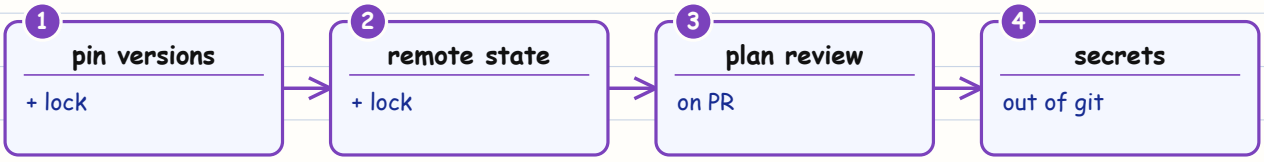
* Code & State

- Pin provider + module versions; commit the lock file; remote state with locking; small blast radius.
- fmt + validate in CI; plan on PR, apply on merge; least-privilege credentials.

* Design

- Use modules for reuse; for_each over count for named sets; keep secrets out of Git; guard stateful resources.

Best practices



CHECKLIST
pinned versions - committed lock file - remote state + locking - plan reviewed - secrets out of git - modules for reuse - prevent_destroy on critical data.

COMMON MISTAKE
Treating infra code more loosely than app code. It deploys real, costly, sometimes irreversible resources. Review, test and version it at least as carefully.

REVISION SNAPSHOT

ASK YOURSELF	Recite the Terraform best-practice checklist.
DO THIS	Audit one of your configs against every checklist item.

35. Terraform Cheat Sheet

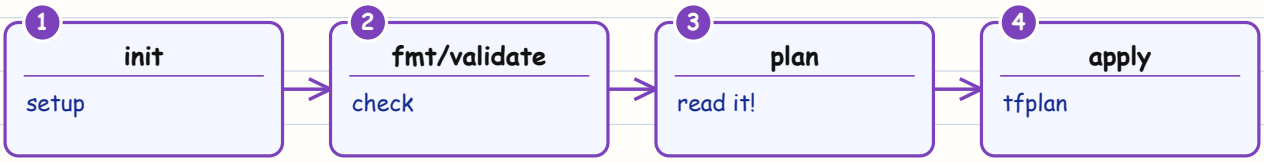
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Terraform commands you will use every day.

* Daily Drivers

- init, fmt, validate, plan -out, apply, destroy, output, state list/show/mv/rm, import, console.

Everyday Terraform



SAFE LOOP
fmt -> validate -> plan (read it!) -> apply. Never skip reading the plan, and never hand-edit state. These two habits prevent most Terraform disasters.

ONE-PAGE COMMAND REFERENCE

```
terraform init            terraform state list
terraform fmt -recursive  terraform state show <addr>
terraform validate        terraform state mv A B
terraform plan -out=tf    terraform state rm <addr>
terraform apply tf        terraform import <addr> <id>
terraform destroy         terraform output -json
terraform console         terraform workspace new/select
terraform graph           TF_LOG=DEBUG terraform apply
```

COMMON MISTAKE
Skipping plan review to save time. The plan is the one moment to catch a destructive replace before it happens. Always read every +/~/-/+ line.

REVISION SNAPSHOT

ASK YOURSELF	What is the safe fmt -> validate -> plan -> apply loop?
DO THIS	Recall ten Terraform commands from memory grouped by purpose.

36. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by provisioning the Shop API's infrastructure the professional way.

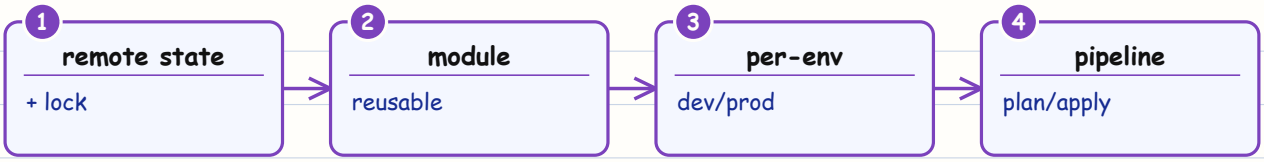
* Provision Shop API Infra

- Write a config with a remote backend + locking, typed variables, and a reusable network module.
- Provision a small stack (e.g. a bucket + security group) with common tags via a local.

* Operate It Properly

- Add outputs for key values, mark secrets sensitive, and separate dev/prod state.
- Wire a CI pipeline that plans on PR and applies on merge.

Capstone steps



REAL-WORLD EXAMPLE

```
terraform init
terraform workspace new dev   # or per-env config
terraform plan -out=tfplan
terraform apply tfplan
terraform output -json
```

STUDENT LAB
Deliverables: remote state + locking, a reusable module, per-env separation, sensitive-marked secrets, outputs for key values, and a plan-on-PR / apply-on-merge pipeline.

REVISION SNAPSHOT

ASK YOURSELF	Is your state remote+locked, modular, and environment-separated?
DO THIS	Complete every capstone deliverable and destroy cleanly at the end.

37. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Declarative vs imperative; what state is and why remote+locking matters; count vs for_each.
- How plan/apply work, how drift happens, and why you never hand-edit state.

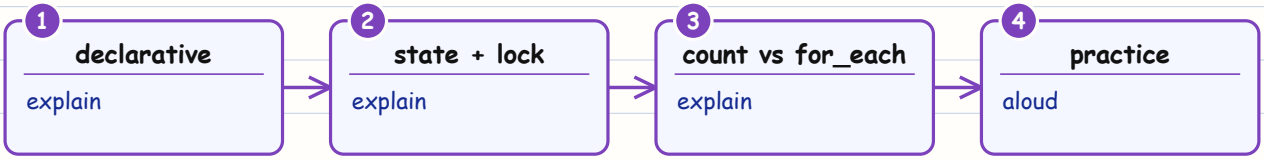
* Common Questions

- How do you manage secrets? How do you structure environments? What does a plan's -/+ mean?
- When do you use modules, data sources, lifecycle rules, and depends_on?

* Your Next Step

- Build the capstone against a free-tier cloud, break it deliberately, detect drift, and recover.

Revise out loud



REAL-WORLD EXAMPLE

```
# 30-second self test
terraform plan          # can you read every +/~/-/+ ?
terraform state list    # what is managed?
# count vs for_each? remote state + locking? drift?
```

REVISION SNAPSHOT

ASK YOURSELF	Can you explain state and remote locking in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.