Terraform

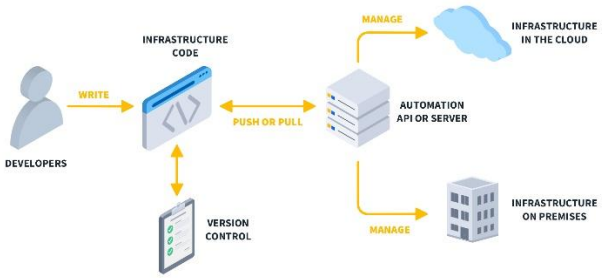
Introduction

What is IT Infrastructure ?

- A traditional IT infrastructure is made up of the usual hardware and software components: facilities, data centers, servers, networking hardware desktop computers and enterprise application software solutions.

What is Infrastructure-as-Code (IaC) ?

- Infrastructure as code involves managing and provisioning data centers through code modules rather than interactive configuration tools or physical hardware configuration.
- In straightforward terms, IaC means managing your IT infrastructure with lines of code.
- The code for the infrastructure becomes part of the project and is stored inside your version control system (VCS).
- IaC works through templates. With IaC, the configuration files are created according to the configuration specifications of your infrastructure. This is simply referred to as **codifying your infrastructure**.



Types of IaC

- **Scripts**:
  - Scripting is a very direct form of IaC.
  - This type of IaC is usually used for simple or one-off tasks and is not advised for complex tasks.
  - **Example:** Bash, PowerShell, Python
- **Configuration management tools**:
  - These tools implement automation by installing and configuring servers.
  - This type of IaC is designed for complex tasks as they are specialized tools built to manage software.
  - **Example:** Ansible, Chef, Puppet, SaltStack
- **Provisioning tools**:
  - These tools implement automation by creating and managing infrastructure (servers, networks, storage).
  - Developers define the end state, and the tool figures out how to achieve it.
  - **Example:** Terraform, CloudFormation, Pulumi
- **Containers and templating tools**:
  - Tools that create pre-configured images (containers, VM templates) with all dependencies to run an application.
  - Data distributed with these tools are easy to manage and have a lower overhead compared to running an entire server.
  - **Example:** Docker, Packer

Agent vs. Agentless Infrastructure Management

Agent-Based Approach

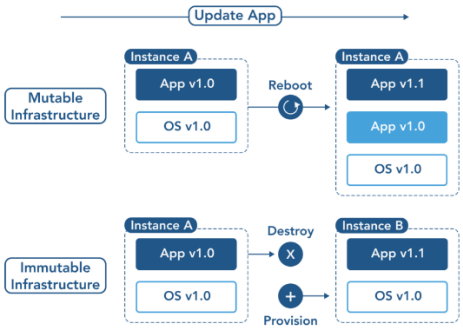
- **Definition**
  - Requires installing a software agent (a small background process) on each managed machine.
  - The agent communicates with a central server to enforce configurations.
- **How It Works**
  - Agent installed on target servers.
  - Central server (master) sends commands to agents.
  - Agent executes tasks (e.g., install software, apply configs).
- **Pros**
  - Persistent control – Agents continuously enforce policies.
  - Efficient for large-scale – Good for managing thousands of servers.
  - Works offline – Agents can cache policies if the central server is unreachable.
- **Cons**
  - Overhead – Agents consume CPU, memory, and bandwidth.
  - Security risks – Agents run with elevated privileges (attack surface).
- **Use Cases**
  - Managing long-lived servers (e.g., data centers, cloud VMs).
  - Enforcing compliance (e.g., security patches, logging).
- **Examples:** Chef (Chef Client), Puppet (Puppet Agent), SaltStack (Salt Minion)

Agentless Approach

- **Definition**
  - No permanent software is installed on managed machines.
  - Uses temporary connections (SSH, WinRM, APIs) to push changes.
- **How It Works**
  - Controller node connects to target machines via SSH/WinRM/API.
  - Executes commands remotely and disconnects.
  - No persistent process remains on the target.
- **Pros**
  - Lightweight – No resource usage when idle.
  - Easier to deploy – No need to install agents.
  - Better security – No long-running privileged processes.
- **Cons**
  - Requires network access – Needs SSH/WinRM/API connectivity.
  - Slower for large fleets – Each connection adds overhead.
  - No continuous enforcement – Only applies changes when triggered.
- **Use Cases**
  - Cloud instances (ephemeral servers, containers).
  - Temporary environments (CI/CD pipelines, auto-scaling groups).
- **Examples:** Ansible (SSH/WinRM), Terraform (Cloud APIs), AWS Systems Manager (SSM)

Mutable vs Immutable Infrastructure

Mutable Infrastructure

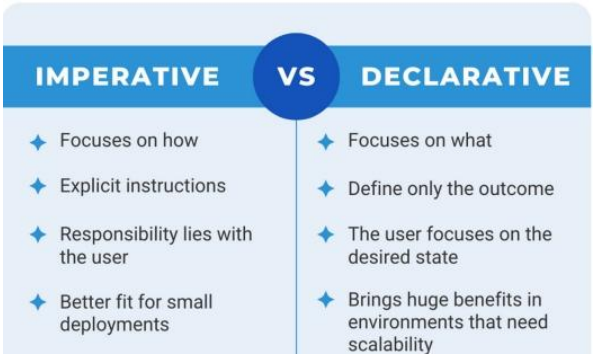


- **Definition**
  - Servers and components can be modified after deployment (e.g., updates, patches, config changes).
  - Traditional IT and some configuration management tools (Ansible, Chef, Puppet) follow this model.
- **How It Works**
  - A server is deployed, then incrementally updated over time.
  - Changes are applied in-place (e.g., apt-get upgrade, editing config files).
- **Pros**
  - Flexibility – Easy to make quick fixes.
  - Familiar – Matches traditional sysadmin workflows.
  - Resource-efficient – No need to reprovision for small changes.
- **Cons**
  - Configuration drift – Servers diverge over time, leading to "snowflake servers."
  - Hard to reproduce – Debugging issues is harder if environments aren't identical.
  - Security risks – Long-lived servers accumulate vulnerabilities.
- **Use Cases**
  - Legacy systems where rebuilding is difficult.
  - Databases (stateful systems where in-place updates are preferred).

Immutable Infrastructure

- **Definition**
  - Servers and components are never modified after deployment.
  - Instead of updating, new instances are deployed from a trusted image, and old ones are discarded.
  - Common in cloud-native and containerized environments (Docker, Kubernetes, Terraform + Packer).
- **How It Works**
  - A golden image (VM/container template) is created with all dependencies.
  - When an update is needed:
    - A new image is built (e.g., with Packer).
    - New servers are deployed from this image.
    - Old servers are terminated.
- **Pros**
  - Consistency – Every deployment is identical.
  - Predictable rollbacks – Revert by deploying the previous image.
  - Security – No long-lived servers with accumulated vulnerabilities.
  - Scalability – Easy to automate in cloud environments.
- **Cons**
  - Slower for small changes – Requires rebuilding and redeploying.
  - Storage overhead – Multiple image versions must be stored.
  - Not ideal for stateful systems (e.g., databases need extra care).
- **Use Cases**
  - Microservices & cloud-native apps (containers, serverless).
  - CI/CD pipelines (fully automated deployments).
  - High-compliance environments (finance, healthcare).

Declarative vs. Imperative Approaches



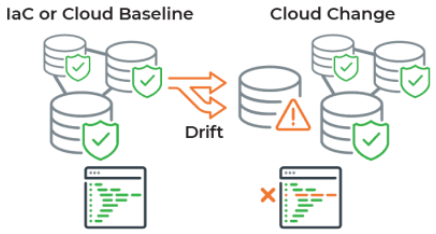
Declarative Approach

- **Definition**
  - You define what the desired end state should be (not how to achieve it).
  - The IaC tool figures out the steps automatically (idempotent execution).
- **How It Works**
  - Write a configuration file describing the final infrastructure state.
  - The tool (e.g., Terraform, AWS CloudFormation) plans and applies changes to match that state.
- **Pros**
  - Idempotent – Running the same script multiple times won't cause errors.
  - Easier to maintain – Focus on the outcome, not implementation details.
  - Better for collaboration – Clear, human-readable intent.
- **Cons**
  - Less control over exact steps – The tool decides execution order.
  - Steeper learning curve – Requires understanding of state management.
- **Tools Using Declarative Style**
  - Terraform
  - AWS CloudFormation
  - Kubernetes (YAML manifests)

Imperative Approach

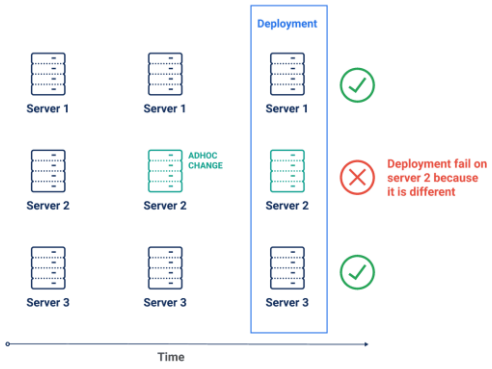
- **Definition**
  - You specify exact steps (commands) to reach the desired state.
  - Similar to scripting (Bash, PowerShell).
- **How It Works**
  - Write a script with sequential commands (e.g., "create VM → install Nginx").
  - The system executes them in order.
- **Pros**
  - Fine-grained control – Useful for complex, custom workflows.
  - Familiar to scripters – Feels like traditional programming.
- **Cons**
  - Not idempotent – Running twice may cause errors (e.g., duplicate resources).
  - Harder to maintain – Must manually handle all edge cases.
- **Tools Using Imperative Style**
  - Ansible (though it has some declarative features)
  - Chef Recipes
  - Shell scripts

Configuration Drifting



- Configuration drift occurs when deployed infrastructure deviates from its intended state due to:
  - Manual changes (e.g., a developer enabling "Delete on Termination" for a production database).
  - Malicious actors (unauthorized modifications).
  - Side effects from APIs, SDKs, or CLI tools.

- **Risks of Unnoticed Drift:**
  - Data breaches or loss.
  - Service interruptions or downtime.
  - Compliance violations.



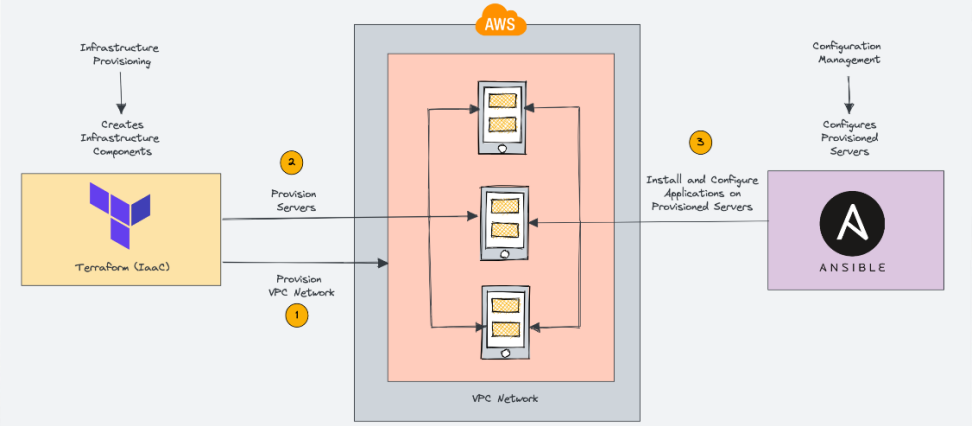
- **How to Detect Configuration Drift?**
  - *Compliance Tools:*
    - AWS Config, Azure Policy, or GCP Security Health Analytics scan for misconfigurations.
  - *Built-in Drift Detection:*
    - AWS CloudFormation compares actual vs. desired state.
  - *IaC State Files:*
    - Terraform state files track expected configurations.
- **How to Correct Configuration Drift?**
  - *Automated Remediation:*
    - Tools like AWS Config auto-correct misconfigurations.
  - *IaC Commands:*
    - `terraform refresh` updates the state file; `terraform plan` identifies drifts.
  - *Manual Fixes (Not Recommended):*
    - Error-prone and inconsistent.
  - *Rebuild Infrastructure:*
    - Destroy and redeploy from IaC templates (immutable approach).
- **How to Prevent Configuration Drift?**
  - *Immutable Infrastructure:*
    - Never modify live servers; replace them entirely (e.g., via Blue/Green deployments).
    - Use pre-baked images (AMI, Containers) built with tools like Packer or AWS Image Builder.
  - *GitOps & IaC:*
    - Version control all infrastructure code in Git.
    - Require peer reviews (Pull Requests) for changes.
  - *Automated Enforcement:*
    - CI/CD pipelines apply changes only through IaC (no manual overrides).

Benefits of Infrastructure-as-Code

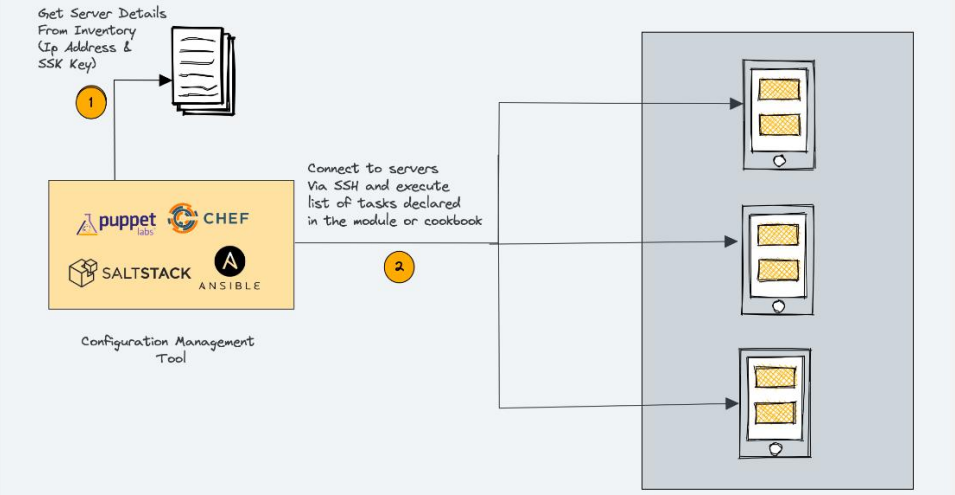
- **Speed & Efficiency**
  - *Faster Provisioning:* Automates infrastructure deployment, reducing setup time from days to minutes.
  - *Consistency:* Eliminates manual errors by ensuring identical environments every time.
  - *Reusability:* Code templates can be reused across projects, saving time and effort.
- **Cost Optimization**
  - *Reduced Manual Labor:* Minimizes the need for sysadmins to manually configure servers
  - *Resource Efficiency:* Auto-scaling and dynamic provisioning prevent over-provisioning.
  - *Pay-as-You-Go:* Cloud resources can be spun up/down based on demand, reducing idle costs.
- **Improved Consistency & Compliance**
  - *Eliminates Configuration Drift:* Ensures all environments (dev, staging, prod) match exactly.
  - *Version Control:* Infrastructure changes are tracked via Git, enabling rollback if needed.
  - *Policy as Code:* Compliance rules (e.g., security, networking) can be embedded in code.
- **Enhanced Collaboration & DevOps Integration**
  - *Team Collaboration:* Infrastructure code can be shared, reviewed, and improved by teams.
  - *CI/CD Integration:* Works seamlessly with DevOps pipelines (e.g., Terraform + Jenkins/GitHub Actions).
  - *Documentation as Code:* IaC files serve as self-documenting infrastructure blueprints.
- **Disaster Recovery & Reliability**
  - *Quick Recovery:* Failed infrastructure can be redeployed rapidly from code.
  - *Immutable Infrastructure:* Instead of patching, new servers are deployed from scratch, reducing inconsistencies.
  - *Disposable Environments:* Ephemeral environments (e.g., for testing) can be created and destroyed easily.
- **Security & Auditing**
  - *Security as Code:* Security policies (firewalls, IAM roles) are defined and enforced via code.
  - *Audit Trails:* All changes are logged, making it easy to track who made changes and why.
  - *Automated Compliance Checks:* Tools like OpenSCAP or Terraform Sentinel enforce security policies.
- **Scalability & Flexibility**
  - *Handles Growth:* Easily scales infrastructure up/down based on demand.
  - *Multi-Cloud Support:* Tools like Terraform, Pulumi, and Ansible work across AWS, Azure, GCP, etc.
  - *Hybrid Cloud:* Manages both cloud and on-premises infrastructure uniformly.

Infrastructure Provisioning vs Configuration Management

- **Provisioning**
  - *Definition:*
    - Provisioning is the process of setting up and allocating infrastructure resources (e.g., servers, networks, storage) so they are ready for use.
  - *Key Actions:*
    - Spinning up virtual machines (VMs), containers, or cloud instances (e.g., AWS EC2, Azure VMs).
    - Creating networks, subnets, load balancers, or databases.
    - Assigning compute/storage resources.
  - *Tools:*
    - Infrastructure-as-Code (IaC): Terraform, AWS CloudFormation, Pulumi.
    - Cloud APIs: AWS CLI, Azure Resource Manager (ARM), Google Cloud SDK.

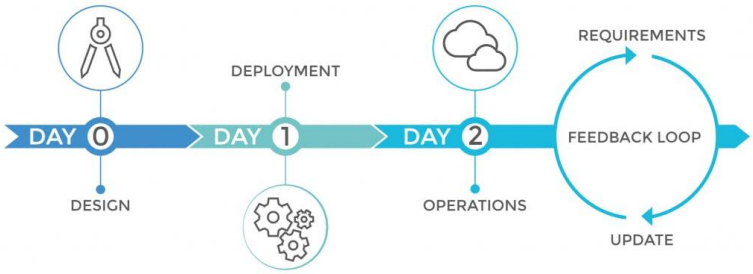


- **Configuration**
  - *Definition:*
    - Configuration is the process of customizing provisioned resources to meet application requirements (e.g., installing software, setting permissions, tuning OS settings).
  - *Key Actions:*
    - Installing dependencies (e.g., Apache, MySQL).
    - Setting up users, permissions, and security policies.
    - Deploying application code or configurations (e.g., nginx.conf).
  - *Tools:*
    - Configuration Management (CM): Ansible, Chef, Puppet, SaltStack.
    - Scripts: Bash, PowerShell, Python.



Note: Infrastructure Provisioning and Configuration Management are not mutually exclusive. Most configuration management tools can do some degree of provisioning and vice versa.

Infrastructure Lifecycle (Day 0, Day 1, Day 2)

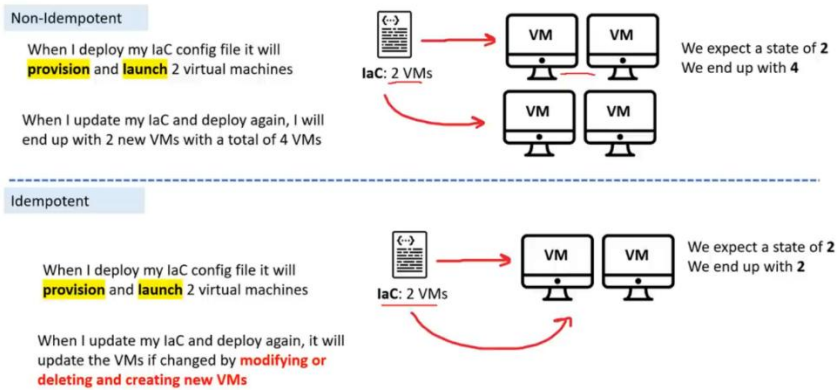


- Infrastructure Lifecycle is a framework used by DevOps engineers to manage cloud infrastructure from planning to retirement.
- It is a structured approach to managing cloud infrastructure, covering:

Planning → Design → Build → Test → Deploy → Maintain → Retire

- It breaks the process into three simplified phases labeled Day 0, Day 1, and Day 2.
- These "days" represent phases (not literal 24-hour periods) in the lifecycle:
- **Day 0: Plan and Design**
    - *Goal:* Define requirements and architect the infrastructure.
    - *Activities:*
        - Assess business/technical needs (e.g., scalability, security).
        - Design cloud architecture (e.g., VPCs, IaC templates).
        - Choose tools (e.g., Terraform, AWS/Azure/GCP services).
    - *Output:* Blueprints, IaC code, compliance policies.
- **Day 1: Develop and Iterate**
    - *Goal:* Build and test the infrastructure.
    - *Activities:*
        - Provision resources using IaC (e.g., Terraform/CloudFormation).
        - Configure software/services (e.g., Ansible, Kubernetes).
        - Test functionality, security, and performance.
    - *Output:* Functional infrastructure ready for deployment.
- **Day 2: Go Live and Maintain**
    - *Goal:* Operate, monitor, and optimize live infrastructure.
    - *Activities:*
        - Deploy to production.
        - Monitor performance (e.g., CloudWatch, Prometheus).
        - Patch/update systems, scale resources, fix drift.
        - Plan for eventual retirement/replacement.
    - *Output:* Stable, efficient, and secure infrastructure.

Idempotent vs Non-idempotent



Idempotent Operations

- **Definition**
  - An operation is idempotent if performing it multiple times has the same effect as performing it once.
- **Key Characteristics**
  - Safe to retry – No side effects if repeated.
  - Consistent state – Ensures the system reaches the desired state regardless of execution count.
  - Essential for automation – Used in IaC, APIs, and configuration management.
- **Examples**
  - *Infrastructure (IaC):*
    - Terraform's apply: Running it multiple times won't create duplicate resources; it ensures the declared state.
    - `aws ec2 create-tags` (tagging an existing resource).
  - *APIs:*
    - `HTTP PUT` (updating a resource with the same data).
    - `HTTP DELETE` (deleting an already-deleted resource has no effect).
  - *Configuration Management:*
    - Ansible playbooks (if a package is already installed, rerunning won't change anything).

Non-Idempotent Operations

- **Definition**
  - An operation is non-idempotent if repeating it produces different results or side effects.
- **Key Characteristics**
  - Unsafe to retry – May cause duplicates, errors, or corruption.
  - State changes with each execution – Each run modifies the system differently.
  - Requires careful handling – Often needs checks or locks to prevent issues.
- **Examples**
  - *Infrastructure (IaC):*
    - Running `aws ec2 run-instances` multiple times creates duplicate VMs.
    - A script that appends data to a file (each run adds more content).
  - *APIs:*
    - `HTTP POST` (creating a new resource; repeating it generates duplicates).
    - A banking API that deducts money (each call withdraws again).
  - *Configuration Management:*
    - A script that increments a counter or appends logs.

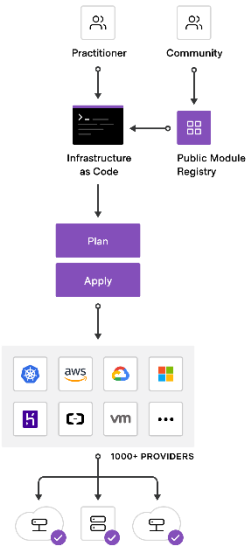
Terraform Basics

What is HashiCorp?

- HashiCorp is a software company that provides open-source and commercial tools for cloud infrastructure automation, focusing on DevOps, security, and multi-cloud workflows.
- Key Products by HashiCorp:
  - Terraform – Infrastructure-as-Code (IaC) tool for provisioning cloud resources.
  - Vault – Secrets management and encryption for applications.
  - Vagrant – Creation of lightweight, reproducible, and portable development environments.
  - Consul – Service mesh and network automation.
  - Nomad – Container and workload orchestration (like Kubernetes but simpler).
  - Packer – Creates machine images (AMIs, Docker images) for immutable infrastructure.
  - Boundary – Secure remote access to systems.
  - Waypoint – Application deployment tool.

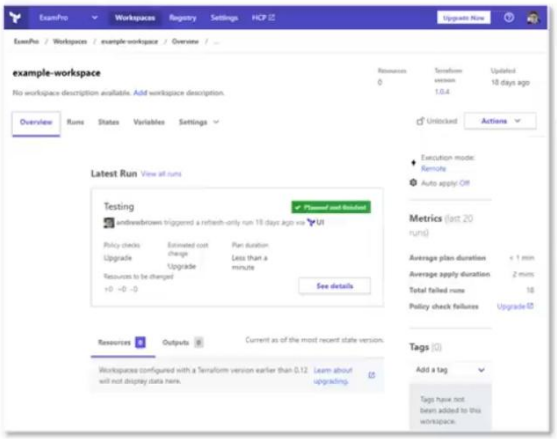
What is Terraform?

- Terraform is an open-source, cloud-agnostic (run seamlessly across any cloud provider) provisioning tool developed by HashiCorp and written in GO language. 28 July 2014.
- Terraform's HCL (HashiCorp Configuration Language) allows a developer to learn one language to work with multiple cloud offerings and on premises providers rather than having to learn a new service and language for each.
- HCL is a declarative language, focusing on the end-state as opposed to a procedural language, where all commands are executed in the order written.
- Terraform is immutable infrastructure and agentless tool.



What is Terraform Cloud?

Terraform Cloud



Terraform Cloud is a Software as Service (SaaS) offering for:

- Remote state storage
- Version Control integrations
- Flexible workflows
- Collaborate on Infrastructure changes

in a single **unified web portal**.

www.terraform.io/cloud

FREE

Terraform Cloud has a **generous free-tier** that allows for team collaboration for the first **5 users** of your organization

In majority of cases you should be using Terraform Cloud.

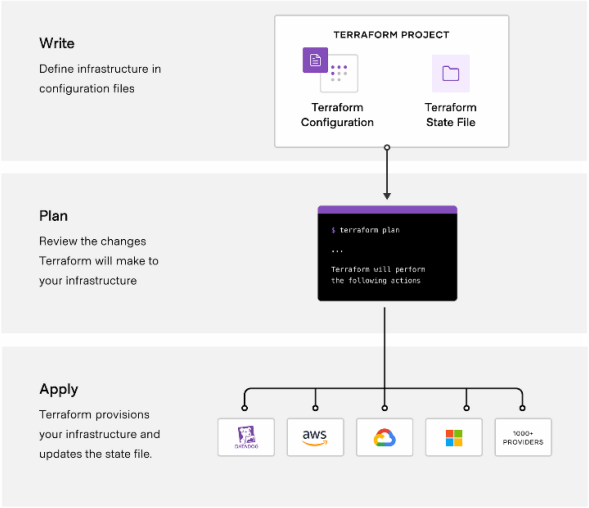
The only case where you may not want to use it to manage your state file is your company has many regulatory requirements along with a long procurement process so you could use Standard remote backend, Atlantis, or investigate Terraform Enterprise

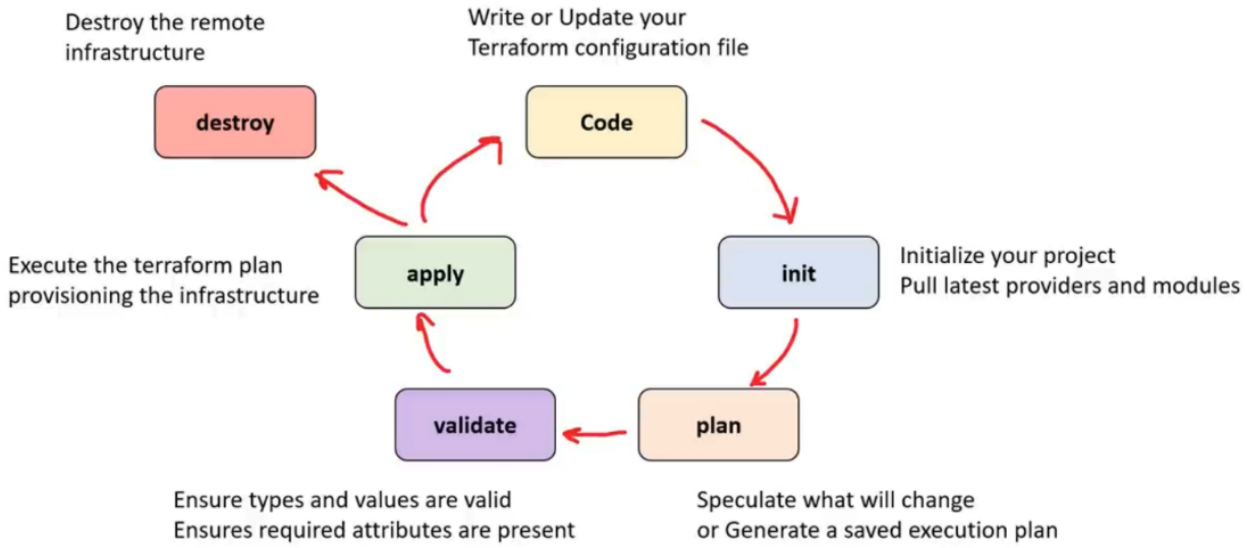
The underlying software for Terraform Cloud and Terraform Enterprise is known as the "Terraform Platform"

Terraform Workflow

The core Terraform workflow consists of three stages:

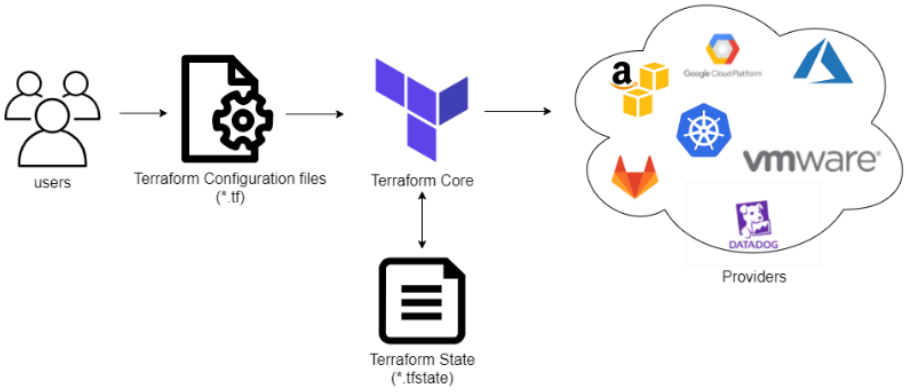
- **Write**: You define resources, which may be across multiple cloud providers and services.
- **Plan**: Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
- **Apply**: On approval, terraform performs the proposed operations in the correct order, respecting any resource dependencies.





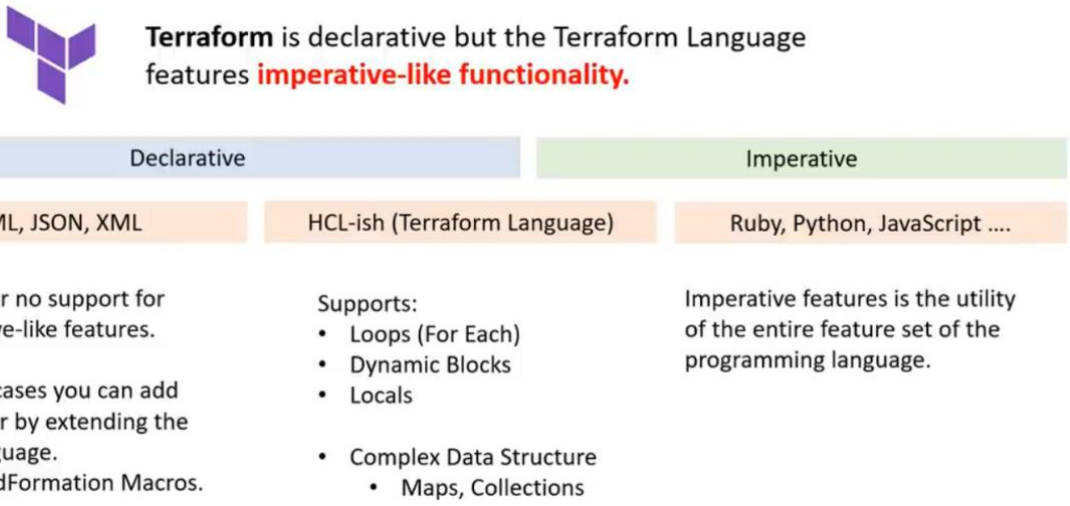
How to deploy infrastructure with Terraform ?

- **Scope** - Identify the infrastructure for your project.
- **Author** - Write the configuration for your infrastructure.
- **Initialize** - Install the plugins Terraform needs to manage the infrastructure.
- **Plan** - Preview the changes Terraform will make to match your configuration.
- **Apply** - Make the planned changes.



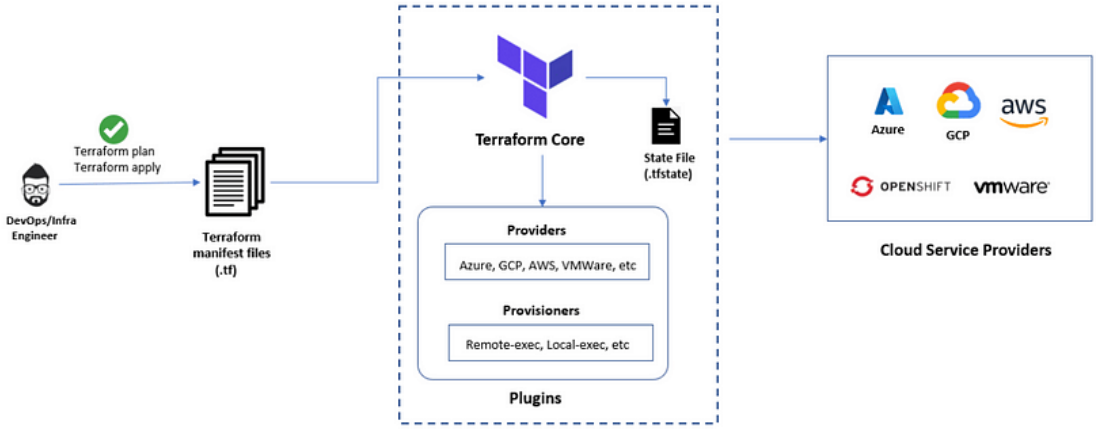
Key features of Terraform

- **Declarative Configuration:** Terraform configurations describe the desired state of your infrastructure, allowing you to specify what resources should be created, their configurations, and their relationships.
- **Infrastructure as Code (IaC):** Terraform treats infrastructure as code, enabling versioning, collaboration, and consistent provisioning across different environments.
- **Multi-Cloud Support:** Terraform supports various cloud providers, such as AWS, Azure, Google Cloud, and others, as well as on-premises solutions. This allows you to manage infrastructure across different cloud platforms with a single set of configuration files.
- **Resource Graph:** Terraform creates a dependency graph of resources based on your configuration, ensuring that resources are created or modified in the correct order.
- **Execution Plans:** Before applying changes to your infrastructure, terraform generates an execution plan that shows what actions will be taken. This helps you understand the impact of changes before they are applied.
- **State Management:** Terraform maintains a state file that records the current state of your infrastructure. This state file is used to plan and apply changes, track resource dependencies, and prevent conflicts in a collaborative environment.
- **Modularity:** Terraform configurations can be modularized, allowing you to reuse and share components across different projects or environments.
- **Community Modules:** The Terraform community provides a wide range of reusable modules for common infrastructure patterns, making it easier to adopt best practices and accelerate development.



Terraform Core vs. Terraform Plugins

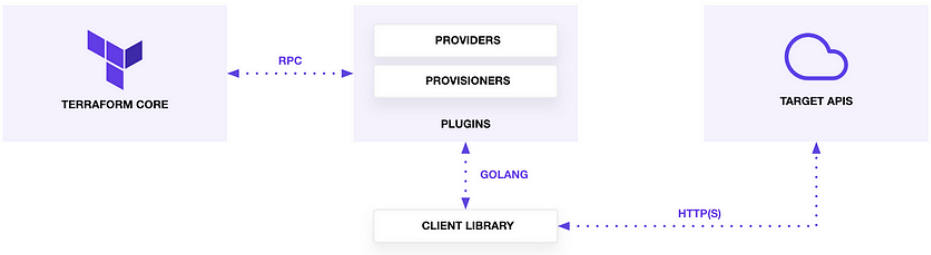
- Terraform's architecture is split into two main components:
  - Terraform Core (The Brain)
  - Terraform Plugins (The Helpers)



Terraform Core

- **What it does:**
  - Manages the workflow (plan, apply, destroy).
  - Reads configuration files (`*.tf`) and constructs a dependency graph.
  - Communicates with plugins to provision resources.
- **Key Responsibilities:**
  - *Parsing Configs:* Understands HCL (HashiCorp Configuration Language).
  - *Dependency Graph:* Determines the order of operations (e.g., create VPC before subnets).
  - *State Management:* Tracks infrastructure state in .tfstate.
  - *Plan/Apply:* Calculates & executes changes safely.
- **Example:**

```
terraform plan  # Core analyzes changes
terraform apply # Core coordinates execution via plugins
```



Terraform Plugins (Providers & Provisioners)

- Plugins extend Terraform's functionality to work with different services.
- **Providers (Cloud/Service Plugins)**
  - Interface with APIs (AWS, Azure, Kubernetes, etc.).
  - Define resource types (e.g., `aws_instance`, `google_storage_bucket`).
  - Example Providers:
    - `hashicorp/aws` (AWS resources)
    - `hashicorp/azurerm` (Azure resources)
    - `hashicorp/kubernetes` (K8s deployments)
  - How They Work:
    - Terraform Core asks the AWS provider: "Create an EC2 instance."
    - The AWS provider translates this into AWS API calls.
  - Declaring a Provider:

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

- **Provisioners (Optional)**
  - Run scripts after resource creation (e.g., local-exec, remote-exec).
  - Used for bootstrapping (e.g., installing software on a new VM).
  - Example:

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo 'Instance ready!' > ip.txt"
  }
}
```

How Core & Plugins Work Together

- User runs `terraform init`
  - Core downloads required plugins (e.g., AWS, Docker).
- User runs `terraform plan`
  - Core checks the state, builds a graph, and asks plugins: "What changes are needed?"
- User runs `terraform apply`
  - Core instructs plugins to make API calls (e.g., aws.ec2.RunInstances).

Types of Terraform Files

- Terraform uses different file types to define, configure, and manage infrastructure.
- Here's a breakdown of the key file types and their purposes:

1. Main Configuration Files (*.tf)

These files contain infrastructure definitions written in HCL (HashiCorp Configuration Language) or JSON.

- `main.tf`
  - Primary file for defining resources, data sources, and modules.
  - **Example:**

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

- `variables.tf`
  - Defines input variables (parameters for your Terraform code).
  - **Example:**

```
variable "region" {
  description = "AWS region"
  default     = "us-east-1"
}
```

- `outputs.tf`
  - Defines output values (to expose information like IPs, IDs).
  - **Example:**

```
output "instance_ip" {
  value = aws_instance.web.private_ip
}
```

- `providers.tf`
  - Configures Terraform providers (AWS, Azure, GCP, etc.).
  - **Example:**

```
provider "aws" {
  region = var.region
}
```

2. Terraform State Files (terraform.tfstate & *.tfstate.backup)

- `terraform.tfstate`
  - JSON file storing the current state of provisioned infrastructure.
  - **Never edit manually!** Managed by Terraform.
- `*.tfstate.backup`
  - Backup of the previous state before changes.
  - *Warning:* State files may contain secrets (passwords, keys). Always:
    - Store them securely (e.g., Terraform Cloud, S3 + DynamoDB).
    - Use `.gitignore` to exclude them from version control.

3. Lock File (.terraform.lock.hcl)

- Records provider/plugin versions to ensure consistency.
- Generated automatically by `terraform init`.

4. Override Files (override.tf or *_override.tf)

- Forcefully override Terraform configurations (rarely used).
- **Example** (`override.tf`):

```
override "aws_instance" "web" {
  instance_type = "t3.micro"  # Forces all instances to t3.micro
}
```

5. Module Files (modules/ Directory)

- Reusable components (e.g., a standardized AWS VPC module).
- Structure:

```
modules/
└── vpc/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

- Called in root config:

```
module "vpc" {
  source = "./modules/vpc"
}
```

6. Temporary & Generated Files

- `.terraform/` (Directory)
  - Stores downloaded providers/modules after `terraform init`.
- `terraform.tfvars` or `*.auto.tfvars`
  - Auto-loaded variable files (alternative to CLI -var flags).
  - **Example** (`terraform.tfvars`):

```
region = "eu-west-1"
```

7. Backend Configuration (backend.tf)

- Defines where Terraform stores state (e.g., S3, Terraform Cloud).
- **Example:**

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "prod/terraform.tfstate"
  }
}
```

8. Ignore File (.terraformignore)

- Lists files/directories Terraform should ignore (like `.gitignore`).
- **Example:**

```
*.tfstate
.terraform/
```

File Type	Purpose
`*.tf`	Main config (HCL) for resources, variables, providers.
`terraform.tfstate`	Tracks real-world infrastructure state.
`.terraform.lock.hcl`	Locks provider versions.
`override.tf`	Forces configuration changes (rare).
`modules/`	Reusable infrastructure components.
`.terraform/`	Cached plugins/modules (from init).
`*.tfvars`	Auto-loaded variable values.
`backend.tf`	Remote state storage (S3, Terraform Cloud).
`.terraformignore`	Excludes files from Terraform operations.

Terraform Registry

- The Terraform Registry is HashiCorp's official marketplace for:
  - Modules (Reusable infrastructure blueprints)
  - Providers (Plugins to manage different services)
  - Policy Packs (Sentinel policies for governance)
- **Types of Registries**
  - *Public Registry (registry.terraform.io)*
    - Free access
    - Hosted by HashiCorp
    - Includes verified modules/providers from AWS, Azure, Google, etc.
    - Example Modules:
      - AWS VPC
      - Azure Kubernetes
  - *Private Registry*
    - Enterprise feature (Terraform Cloud/Enterprise)
    - Host internal modules
    - Control access via SSO/RBAC

Terraform Language

HashiCorp Configuration Language (HCL)

HCL is an open-source toolkit for creating **structured configuration languages** that are both human and machine friendly, for use with command-line tools
github.com/hashicorp/hcl

HashiCorp Configuration Language (HCL)

- Terraform Language (.tf) — eg. Dynamic Blocks, For Each…
- Packer Template (.pkr.hcl)
- Vault Policies (no extension)
- Boundary Controllers and Workers (.hcl)
- Consul Configuration (.hcl)
- Waypoint Application Configuration(.hcl)
- Nomad Job Specifications (.nomad)
- Shipyard Blueprint (.hcl)
- Sentinel Policies ✗ ← Doesn't use HCL but its own ACL custom language

Terraform Settings

The special **terraform configuration block type** eg. **terraform { … }** is used to configure some behaviors of Terraform itself

In Terraform settings we can specify:

- **required_version**
  - The expected version of terraform
- **required_providers**
  - The providers that will be pull during an terraform init
- **experiments**
  - Experimental language features, that the community can try and provide feedback
- **provider_meta**
  - module-specific information for providers

```
terraform {
  required_providers {
    aws = {
      version = ">= 2.7.0"
      source = "hashicorp/aws"
    }
  }
}
```

Terraform Language

Terraform language consists of only a few basic elements:

- **Blocks —** containers for other content, represent an object
  - block type — can have zero or more labels and a body
  - block label — name of a block
- **Arguments —** assign a value to a name
  - They appear within blocks
- **Expressions —** represent a value, either literally or by referencing and combining other values
  - They appear as values for arguments, or within other expressions.

```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Values, Variables and Outputs

The Terraform language includes a few kinds of blocks for requesting or publishing named values.

- **Input Variables** serve as parameters for a Terraform module, so users can customize behavior without editing the source.
- **Output Values** are like return values for a Terraform module.
- **Local Values** are a convenient feature for assigning a short name to an expression.

Value Types

- Terraform uses a type system to validate configurations before execution.
- Understanding value types is essential for writing correct, maintainable infrastructure code.

Primitive Types (Basic Types)

Type	Description	Example
string	Text values (UTF-8)	"us-east-1", "production"
number	Numeric values (int/float)	42, 3.14159
bool	Boolean values	true, false

Examples:

```
variable "region" {
  type    = string
  default = "us-west-2"
}

variable "node_count" {
  type    = number
  default = 3
}

variable "enable_ssl" {
  type    = bool
  default = true
}
```

Complex Types

- A complex type is a type that groups multiple values into a single value.
- These values could be of a similar type (collection) or different types (structural).

Collection Types

- A collection allows multiple values of one other type to be grouped together as a single value.
- The type of value Within a collection is called its element type.

Type	Description	Example
`list(T)`	Ordered sequence of type T	`["a", "b", "c"]`
`set(T)`	Unordered unique values of type T	`["web", "db"]` (duplicates removed)
`map(T)`	Key-value pairs (keys are strings)	`{ env = "prod", tier = "frontend" }`

Note: set is similar to list, but it has no index, no preserved order, all values must be the same type and will be cast to match the first element.

Examples:

```
variable "availability_zones" {
  type    = list(string)
  default = ["us-west-2a", "us-west-2b"]
}

variable "tags" {
  type    = map(string)
  default = {
    Environment = "Production"
    Owner       = "DevOps"
  }
}
```

Structural Types

- A structural type allows multiple values of several distinct types to be grouped together as a single value.
- Structural require a schema as an argument to specify Which are allowed for Which elements.

Type	Description	Example
object({...})	Fixed schema with named attributes	{ name = string, ports = list(number) }
tuple([...])	Fixed-length sequence with positional types	[string, number, bool]

Examples:

```
variable "database" {
  type = object({
    name     = string
    engine   = string
    port     = number
    replicas = optional(number, 1)  # Terraform 1.3+
  })
  default = {
    name     = "postgres"
    engine   = "PostgreSQL"
    port     = 5432
  }
}
```

Type Conversions

```
# String → Number (implicit)
resource "aws_instance" "example" {
  instance_type = "t2.micro"  # String works where number expected
  count         = "3"         # String → Number conversion
}

# Explicit conversion functions
locals {
  str_to_num = tonumber("42")       # 42 (number)
  num_to_str = tostring(42)         # "42" (string)
  bool_to_str = tostring(true)      # "true" (string)
}
```

Optional Modifier

- In Terraform, `optional` is a modifier used within object type definitions (especially inside `variable` blocks or `module` input variables) to mark a specific attribute as not required when passing input values.
- **By default**, all object attributes are required. With `optional()`, you can omit the field when calling the `module` or using the `variable`.
- `optional` is used in:
    - Variable Definitions (in `variable` blocks with type constraints).
    - Output Definitions (in `output` blocks with complex types).
    - Type Constraints (when using `object({...})` type expressions).
- If an `optional` attribute is not provided, Terraform sets it to `null`.
- You can also use `optional()` with a default value.
    - If the attribute is omitted, it will use "`default_value`" instead of `null`.

```
optional(string, "default_value")
```

- **Examples:**

```
variable "server" {
  type = object({
    name    = string
    cpu     = optional(number, 2)  # Defaults to 2 if not provided
    memory  = optional(string)     # Defaults to null if not provided
  })
}

# Usage in `terraform.tfvars`:
# server = {
#   name = "web-server"
#   # cpu and memory are optional
# }
```

```
variable "example" {
  type = object({
    required_attr = string
    optional_attr = optional(string)  # This attribute is not required
  })
  default = {
    required_attr = "must_be_provided"
    # optional_attr can be omitted
  }
}
```

Input Variables

- These are defined in your configuration files using the variable block.
- Input variables can have default values, types, and descriptions.
- They are typically used to represent values that vary between environments or configurations, such as instance sizes, region names, or API keys.

Input variables (aka variables or Terraform Variables)
are **parameters** for Terraform modules

Variables are defined via **variable blocks.**

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

You can declare variables in either:

- The root module
- The child modules

Default	A default value which then makes the variable optional
type	This argument specifies what value types are accepted for the variable
Description	This specifies the input variable's documentation
Validation	A block to define validation rules, usually in addition to type constraints
Sensitive	Limits Terraform UI output when the variable is used in configuration

Environment Variables

- Environment variables in Terraform are variables that are set within the environment in which Terraform is executed.
- They provide a way to pass dynamic values or sensitive information to Terraform configurations without hardcoding them directly into the configuration files.
- **Naming Convention:**
  - Environment variables in Terraform typically follow a naming convention where the variable name begins with `TF_VAR_`
  - This convention helps Terraform recognize environment variables as input variables.
- **Setting Environment Variables:**
  - You can set environment variables using various methods, such as exporting them in your shell configuration files (e.g., .bashrc, .bash_profile), setting them temporarily in your terminal session, or configuring them in your CI/CD pipeline environment.
  - An example is the following bash command:

```
export TF_VAR_region="us-west-2"
```

- **Usage in Terraform Configuration:**
  - Once environment variables are set, you can reference them in your Terraform configuration files using the var function.

```
provider "aws" {
  region = var.region
}
```

Loading Input Variables (Precedence)

Default Autoloaded Variables file terraform.tfvars	When you create a named **terraform.tfvars** file it will be automatically loaded when running terraform apply
Additional Variables Files (not autoloaded) my_variables.tfvars	You can create additional variables files eg. dev.tfvars, prod.tfvars They will not be autoloaded (you'll need to specific them in via command line)
Additional Variables Files (autoloaded) my_variables.**auto**.tfvars	If you name your file with auto.tfvars it will always be loaded
Specify a Variables file via Command Line **-var-file** dev.tfvars	You can specific variables inline via the command line for individual overrides
Inline Variables via Command Line **-var** ec2_type="t2.medium"	You can specific variables inline via the command line for individual overrides
Environment Variables **TF_VAR_**my_variable_name	Terraform will watch for environment variables that begin with TF_VAR_ and apply those as variables

The definition precedence is the order in which Terraform will read variables and as it goes down the list it will override variable.

- Environment Variables
- terraform.tfvars
- terraform.tfvars.json
- *.auto.tfvars or *.auto.tfvars.json
- -var and -var-file

Local Values

- Local values (defined with locals) are named expressions that simplify complex Terraform configurations by:
  - Reducing repetition
  - Improving readability
  - Centralizing logic
- **Key Characteristics**
  - *Scoped Locally*
    - Only available within the module where defined
  - *Evaluated Lazily*
    - Computed when referenced, not at declaration
  - *Immutable*
    - Cannot be modified after definition
  - *Type Flexible*
    - Can hold any Terraform type (`string`, `list`, `object`, etc.)
- You can define multiple locals blocks and you can reference one into another.
- Once you define a local value, you can reference it by using `local.<value>`
- **Example:**

```
locals {
  # Simple value
  environment = "prod"

  # Computed value
  name_prefix = "${var.project_name}-${local.environment}"

  # Complex structure
  tags = {
    Owner       = "DevOps"
    Environment = local.environment
  }
}
```

Outputs

- Outputs in Terraform expose information about your infrastructure for:
  - Display in the CLI
  - Sharing between modules
  - Integration with other systems (CI/CD, scripts)
- They provide a way to retrieve useful information about the resources created by Terraform, such as IP addresses, DNS names, or ARNs.
- **Key Characteristics**
  - *Explicit Exposure*
    - Makes selected values available to parent modules/users
  - *Post-Apply Visibility*
    - Only shown after `terraform apply`
  - *Read-Only*
    - Cannot modify infrastructure (only query)
  - *Type Constraints*
    - Can specify exact output types
- You can mark an output as a sensitive so it will not show up in your terminal output, but it still can be read from the `statefile`.
- **Example:**

```
output "db_password" {
  value       = aws_db_instance.main.password
  sensitive   = true  # Hides value in CLI
}
```

- After applying your Terraform configuration, you can use the `terraform output` command to view the values of defined outputs.

```
terraform output           # All outputs
terraform output vpc_id    # Specific output
```

Data Sources

- Data sources in Terraform allow you to fetch and reference information from external systems without managing them as part of your infrastructure.
- They provide read-only access to existing resources or external data.
- **Key Characteristics**
  - *Read-Only*
    - Cannot modify resources
  - *Dynamic Lookup*
    - Fetches current state during terraform apply
  - *Provider-Specific*
    - Each provider offers its own data sources
  - *No Cost*
    - Doesn't create or manage infrastructure
- **Common Use Cases**
  - Querying Existing Infrastructure

```
data "aws_vpc" "default" {
  default = true
}
resource "aws_subnet" "example" {
  vpc_id     = data.aws_vpc.default.id
  cidr_block = "10.0.1.0/24"
}
```

- Fetching AMI IDs

```
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners      = ["amazon"]

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

- الfilter هنا مخصوص بس لaws وتحديدا ami كمان يعنى مثلا مش موجود في rds.

- Reading Files/Templates

```
data "template_file" "user_data" {
  template = file("${path.module}/user_data.sh")
  vars = {
    db_host = aws_db_instance.main.address
  }
}
```

- **Full example:**

```
# Get existing VPC
data "aws_vpc" "main" {
  tags = {
    Name = "production"
  }
}

# Get latest Ubuntu AMI
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

# Create instance using fetched data
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
  subnet_id     = data.aws_vpc.main.public_subnets[0]

  tags = {
    Name = "web-${data.aws_vpc.main.id}"
  }
}

output "instance_details" {
  value = {
    ami_id    = data.aws_ami.ubuntu.id
    vpc_cidr  = data.aws_vpc.main.cidr_block
    public_ip = aws_instance.web.public_ip
  }
}
```

Resource Meta-Arguments

- Meta-arguments are special arguments that control how Terraform manages resources at a fundamental level.
- They work across all resource types and providers.

depends_on

The order of which resources are provisioned is important when resources depend on others before they are provisioned.

Terraform implicitly can determine the order of provision for resources but there may be some cases where it cannot determine the correct order.

depends_on allows you to explicitly specific a dependency of a resource.

```
resource "aws_iam_role_policy" "example" {
  name   = "example"
  role   = aws_iam_role.example.name
  # ...
}

resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
  iam_instance_profile = aws_iam_instance_profile.example
  depends_on = [
    aws_iam_role_policy.example,
  ]
}
```

count

When you are managing a pool of objects eg. a fleet of Virtual Machines you can use **count**.

Specify the **amount** of instances you want

Get the current count **value** (index) via **count.index**

This value starts at 0

```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

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

Count can accept **numeric expressions**:

- Must be whole number
- Number must be known before configuration

```
resource "aws_instance" "server" {
  # Create one instance for each subnet
  count = length(var.subnet_ids)
  # ....
```

- Access individual instances via `aws_instance.server[0]`.

for_each

- In Terraform, `for_each` is a meta-argument used to create multiple instances of a resource or module based on a collection.
- The `for_each` meta-argument accepts a map or a set of strings and creates an instance for each item in that map or set.
- Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

- This provides the flexibility of dynamically setting the attributes of each resource instance created.
- In blocks where `for_each` is set, an additional `each` object is available in expressions, so you can modify the configuration of each instance. This object has two attributes:
  - `each.key` — The map key (or set member) corresponding to this instance.
  - `each.value` — The map value corresponding to this instance. (If a set was provided, this is the same as `each.key`)

Examples

- Using a Set of Strings

```
resource "aws_instance" "server" {
  for_each = toset(["web", "db", "cache"])  # Convert list to set

  ami           = "ami-123456"
  instance_type = "t3.micro"
  tags = {
    Name = each.key  # "web", "db", "cache"
  }
}
```

- Using a Map

```
resource "aws_iam_user" "users" {
  for_each = {
    "alice" = "admin"
    "bob"   = "developer"
    "eve"   = "readonly"
  }

  name = each.key       # "alice", "bob", "eve"
  tags = {
    Role = each.value   # "admin", "developer", "readonly"
  }
}
```

Accessing Created Resources

- Individual Instances

```
# Reference specific instance by key
output "web_server_id" {
  value = aws_instance.server["web"].id
}
```

- All Instances

```
# Output all instances as a map
output "all_buckets" {
  value = {
    for name, bucket in aws_s3_bucket.data :
    name => bucket.arn
  }
}
```

Provider

- Specifies a non-default provider configuration.

```
provider "aws" {
  alias  = "europe"
```

```
  region = "eu-west-1"
}

resource "aws_instance" "eu_server" {
  provider   = aws.europe  # Uses the european config
  ami        = "ami-789012"
  # ...
}
```

Lifecycle

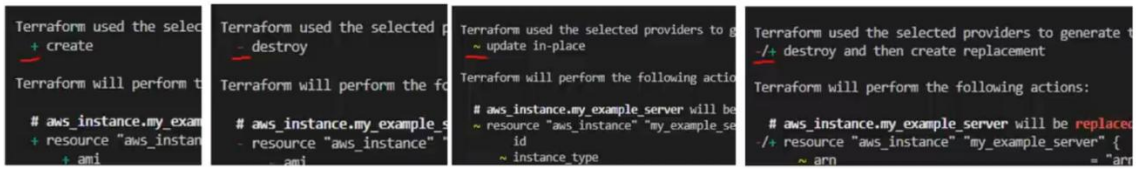
When you execute an execution order via Terraform Apply it will perform one of the following to a resource:

Create
- resources that exist in the configuration but are not associated with a real infrastructure object in the state.

Destroy
- resources that exist in the state but no longer exist in the configuration.

Update in-place
- resources whose arguments have changed.

Destroy and re-create
- resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.



- The lifecycle meta-argument controls how Terraform manages resource creation, updates, and destruction.
- It's a powerful tool for handling special cases in your infrastructure management.
- **Lifecycle Options**
  - `create_before_destroy` (Zero-Downtime Updates)
    - Ensures new resources are created before old ones are destroyed.
    - Use Cases:
      - Databases
      - Load balancers
      - Stateful applications

```
resource "aws_instance" "web" {
  # ...

  lifecycle {
    create_before_destroy = true  # Critical for stateful services
  }
}
```

  - `prevent_destroy` (Safety Net)
    - Blocks accidental destruction.
    - Use Cases:
      - Production databases
      - Critical network infrastructure

```
resource "aws_db_instance" "production" {
  # ...

  lifecycle {
    prevent_destroy = true  # Must disable before terraform destroy
  }
}
```

- `ignore_changes` (Selective Updates)
  - Prevents updates to specific attributes.
  - Commonly Ignored Attributes:
    - Auto-generated tags
    - Timestamps
    - Automatically scaling values

```
resource "aws_autoscaling_group" "web" {
  # ...

  lifecycle {
    ignore_changes = [
      desired_capacity,  # Ignore ASG size changes
      target_group_arns  # Ignore LB attachment changes
    ]
  }
}
```

- `replace_triggered_by` (Controlled Replacement)
  - Forces resource replacement when referenced resources change.

```
resource "aws_instance" "app" {
  # ...

  lifecycle {
    replace_triggered_by = [
      aws_security_group.app.id  # Recreate if SG changes
    ]
  }
}
```

Expressions

- Expressions in Terraform allow you to compute values dynamically within your configuration.
- They're used to reference values, transform data, and make decisions.

Primitive Expressions

- Basic value literals:

```
string = "hello"       # String
number = 42            # Number
bool   = true          # Boolean
tags   = {env = "Production", priority = 3} # Map(object)
null   = null          # Null value
```

Reference Expressions

- Access values from elsewhere:

```
aws_instance.web.id            # Resource attribute
var.region                     # Input variable
module.vpc.vpc_id              # Module output
data.aws_ami.ubuntu.id         # Data source
```

Operators

Perform calculations and comparisons:

Type	Operators	Example
Arithmetic	`+ - * / %`	`var.instances * 2`
Comparison	`== != > < >= <=`	`var.env == "prod"`
Logical	`&& || !`	`!var.test_mode && var.secure`
Conditional	`? :`	`var.env == "prod" ? "large" : "small"`

Function Calls

- Built-in transformations:

```
upper("hello")                 # "HELLO"
length(["a", "b"])             # 2
file("${path.module}/script.sh")
```

for Expressions

- `for` expressions create new collections by transforming existing ones, similar to list comprehensions in Python.
- **Basic Syntax**

```
[for item in LIST : TRANSFORM_EXPRESSION]
{for key, value in MAP : NEW_KEY => NEW_VALUE}
```

    - Working with Lists/Sets

```
# Simple transformation
upper_names = [for name in ["alice", "bob"] : upper(name)]  # ["ALICE", "BOB"]
```

```
# With index
indexed = [for i, name in ["alice", "bob"] : "${i}-${name}"]  # ["0-alice", "1-bob"]

# Conditional filtering
even_numbers = [for n in [1, 2, 3, 4] : n if n % 2 == 0]  # [2, 4]
```

- Working with Maps

```
# Transform keys and values
lowercase_tags = {for k, v in {"Name" = "Web", "Env" = "Prod"} : lower(k) => lower(v)}
# {"name" = "web", "env" = "prod"}

# Filter map entries
production_only = {for k, v in var.instances : k => v if v.env == "prod"}
```

Advanced Examples

```
# Create a map of instance IDs to AZs
instance_azs = {for inst in aws_instance.web : inst.id => inst.availability_zone}

# Flatten nested structures
all_ips = flatten([
  for subnet in aws_subnet.public : subnet.cidr_block
])
```

Splat Expression

A **splat** expression provides a **shorter expression** for **for expressions**

What is a splat operator?
A splat operator is represented by an asterisk (*), it originates from the ruby language
Splats in Terraform are used to rollup or soak up a bunch of iterations in a *for expression*

For lists, sets and tuples

```
[for o in var.list : o.id]
[for o in var.list : o.interfaces[0].name]
```

→

Can be written list this

```
var.list[*].id
var.list[*].interfaces[0].name
```

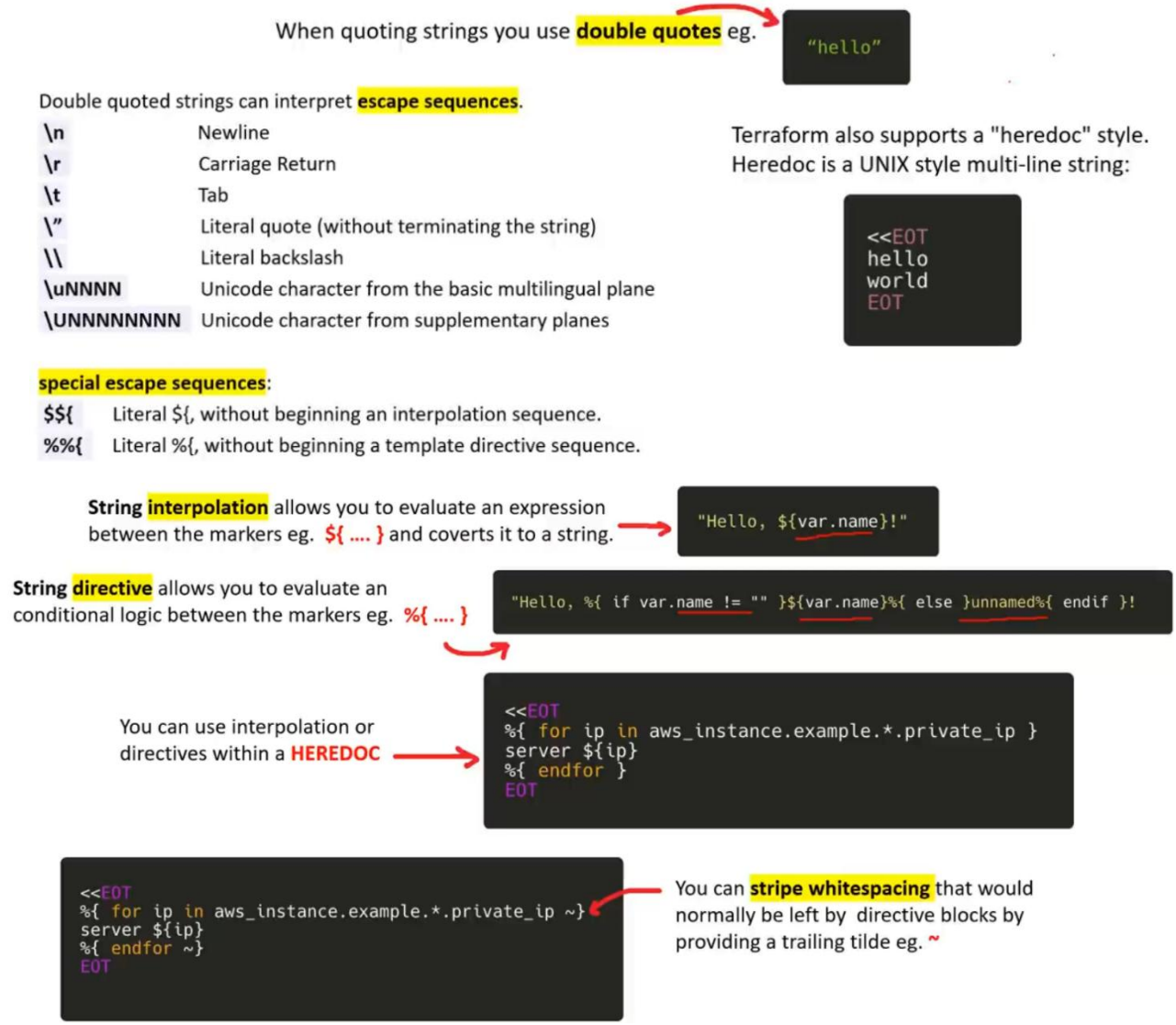
Splat expressions have a special behavior when you apply them to **lists**

- If the value is anything other than a null value then the splat expression will transform it into a single-element list
- If the value is *null* then the splat expression will return an empty tuple.

This behavior is useful for modules that accept optional input variables whose default value is null to represent the absence of any value to adapt the variable value to work with other Terraform language features that are designed to work with collections.

```
variable "website" {
  type = object({
    index_document = string
    error_document = string
  })
  default = null
}

resource "aws_s3_bucket" "example" {
  # ...

  dynamic "website" {
    for_each = var.website[*]
    content {
      index_document = website.value.index_document
      error_document = website.value.error_document
    }
  }
}
```

Strings and Strings Templates



Dynamic Blocks

- Dynamic blocks in Terraform provide a way to generate repetitive nested configurations dynamically based on input data structures.
- They allow you to iterate over a list or map of values and generate multiple instances of a nested block within your Terraform configuration.
- Dynamic blocks are particularly useful when you need to define multiple similar configurations **within a single resource or module**, such as multiple security group rules, multiple AWS S3 bucket policies, or multiple Kubernetes resource definitions.
- **Syntax:**

```
resource "some_resource" "example" {
  # Regular attributes

  dynamic "BLOCK_TYPE" {
    for_each = COLLECTION  # List, set, or map
    content {
      # Block configuration
      ATTRIBUTE = dynamic.value.key  # or .value
    }
  }
}
```

- **Examples:**
    - Security Group Rules

```
resource "aws_security_group" "example" {
  name = "dynamic_ports"

  dynamic "ingress" {
    for_each = var.open_ports
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

    - Nested Configurations

```
resource "aws_autoscaling_group" "example" {
  dynamic "tag" {
    for_each = var.tags
    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

    - Custom Iterator Names

```
dynamic "ingress" {
  for_each = var.security_rules
  iterator = "rule"  # Changes reference from ingress.value to rule.value
  content {
    from_port   = rule.value.from_port
    to_port     = rule.value.to_port
    protocol    = rule.value.protocol
  }
}
```

    - Conditional Blocks

```
dynamic "logging" {
  for_each = var.enable_logging ? [1] : []
  content {
    bucket = aws_s3_bucket.logs.bucket
  }
}
```

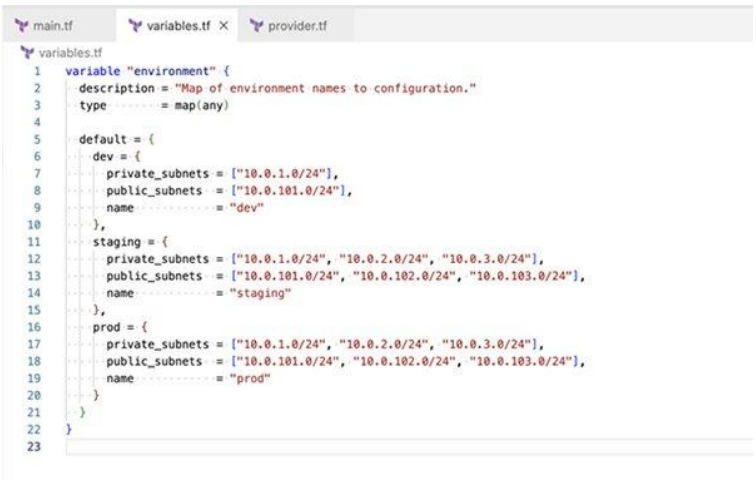
Dynamic blocks vs for_each

- Passing the `for_each` argument to a cloud resource tells Terraform to create a different resource for each item in a map or list.
- **Example: Using for_each to create a VPC resource**
    - To illustrate, use a Terraform module to create a simple virtual private cloud (VPC) resource in AWS, as shown in Figure 1.

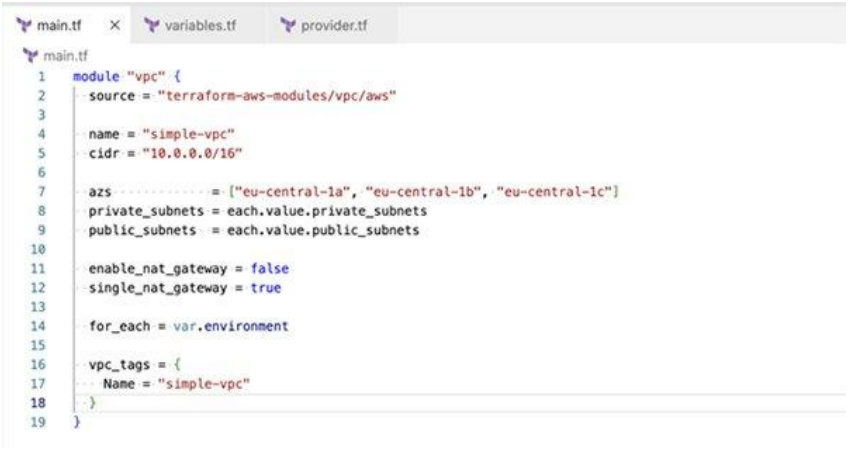
main.tf

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "simple-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["eu-west-1a", "eu-west-1b", "eu-west-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = false
  single_nat_gateway = true


  vpc_tags = {
    Name = "simple-vpc"
  }
}
```

- To create the same resource for multiple environments, start by adding a map of the different environments to create along with the values to be used for each environment, as shown in Figure 2.



- The environment variable is a map of different maps, which you can then reference in the main Terraform file, as shown in Figure 3.



- Adding the `for_each` attribute on line 14 and setting its value to the environment variable tells Terraform to create a VPC module for each item in the environment variable map.
- Lines 8 and 9 reference the private and public subnet lists contained in the map of each environment using the syntax `each.value.<list_name>`.
- In other words, for each item in the environment map, get the value of that item and then get the item by name.

- يبقى عندى الfor_each دة عبارة عن attribute بشغله جوة الresource او جوة الmodule اللى بيعمل الresource عشان يعمل ليا اكتر من instance للresource دة كله على بعضه بناء على map او list انا بديها ليها عشان يمشى عليها ، يعنى انا بيكون عندى resource واحد وباستخدام الfor_each بخلق منه نسخ كتير.

- In Terraform, dynamic blocks let you create nested blocks inside a resource based on a variable. Instead of creating a resource for each item in a map, as the `for_each` attribute does, dynamic blocks create nested blocks inside a resource for each item in a map or list.
- **Example: Using dynamic blocks to simplify ingress and egress rules**
    - Figure 4 shows an AWS security group resource. This security group has a few ingress and egress rules to control access to another resource within the VPCs created in the `for_each` example prior.

main.tf × variables.tf provider.tf

main.tf

```
resource "aws_security_group" "simple_sg" {
  name        = "simple_sg"
  description = "Simple security group"
  vpc_id      = module.vpc[each.key].vpc_id

  for_each = var.environment

  ingress {
    protocol    = "tcp"
    from_port   = 80
    to_port     = 80
    cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
  }
  ingress {
    protocol    = "tcp"
    from_port   = 443
    to_port     = 443
    cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
  }
  egress {
    protocol    = "tcp"
    from_port   = 80
    to_port     = 80
    cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
  }
  egress {
    protocol    = "tcp"
    from_port   = 443
    to_port     = 448
    cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
  }
  egress {
    protocol    = "tcp"
    from_port   = 5432
    to_port     = 5432
    cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
  }
}
```

    - As Figure 4 shows, there's quite a bit of duplication, which means an opportunity to clean up this repetition with Terraform's dynamic blocks.
    - After converting the ingress and egress rules to dynamic blocks, the Terraform code should look like Figure 5.

main.tf × variables.tf provider.tf

main.tf

```
resource "aws_security_group" "simple_sg" {
  name        = "simple_sg"
  description = "Allow TLS inbound traffic"
  vpc_id      = module.vpc[each.key].vpc_id

  for_each = var.environment

  dynamic "ingress" {
    for_each = var.ingress
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
    }
  }

  dynamic "egress" {
    for_each = var.egress
    content {
      from_port   = egress.value
      to_port     = egress.value
      protocol    = "tcp"
      cidr_blocks = [module.vpc[each.key].vpc_cidr_block]
    }
  }
}
```

    - The dynamic blocks contain the `for_each` attribute, which is assigned to different variables now. Figure 6 shows the definitions of the variables used.

main.tf variables.tf × provider.tf

variables.tf

```
variable "ingress" {
  type        = list(string)
  default     = [80, 443]
  description = "open incoming ports for security group"
}

variable "egress" {
  type        = list(string)
  default     = [80, 443, 5432]
  description = "open outgoing ports for security group"
}
```

- Note that the name of the dynamic block is not simply a string. It must match the name of the final block you intend to create.
- Next, the dynamic block creates final blocks based on the values in the map or list specified by the `for_each` attribute. In concept this is very similar to the `for_each` attribute acting on a resource.
- Each value from the map or list is used to define values in the content of the final ingress or egress block.
- If the map or list is empty, terraform does not create any final block.
- This feature can be useful when implemented as a mechanism to disable or enable certain features of a resource controlled by blocks defined in that resource.

- أما بقى الdynamic blocks فدى بستخدمها عشان اعمل nested blocks جوة الresource الواحد اللى انا بعمله ، زى مثلا لما اجى اعمل security group دة كدة resource واحد ، طب جواه بقى انا عاوز اعمل اكتر من ingress rule يبقى ساعتها استخدم الdynamic blocks.

- الاتنين بقى متشابهين مع بعض في انهم بيعتمدوا على variable input بيدخل ليهم عشان يشتغلوا على أساسه ، يعنى الfor_each محتاجة variable input عشان تعمل كذا resource منه ، والdynamic blocks محتاجة برضو variable input عشان تعمل كذا block جوة الresource على أساسه.

Version Constraints

Terraform utilizes Semantic Versioning for specifying Terraform, Providers and Modules versions

Semantic Versioning is **open-standard** on how to define versioning for software management e.g. **MAJOR.MINOR.PATCH**

2.0.0 2.0.0-rc.2 2.0.0-rc.1 1.0.0 1.0.0-beta

semver.org

1.MAJOR version when you make incompatible API changes,
2.MINOR version when you add functionality in a backwards compatible manner, and
3.PATCH version when you make backwards compatible bug fixes.
Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

A **version constraint** is a string containing one or more conditions, separated by commas.

- = or no operator. Match exact version number e.g. "1.0.0", "=1.0.0"
- != Excludes an exact version number e.g. "!=1.0.0"
- > >= < <= Compare against a specific version e.g. ">= 1.0.0"
- ~> Allow only the rightmost version (last number) to increment e.g. ~> 1.0.0"

Buit-in Functions

Terraform language includes several built-in functions that you can call from within expressions to transform and combine values.

Numeric functions

abs returns the absolute value of the given number

```
> abs(23)
23
> abs(0)
0
> abs(-12.4)
12.4
```

ceil returns the closest whole number that is greater than or equal to the given value

```
> ceil(5)
5
> ceil(5.1)
6
```

floor returns the closest whole number that is less than or equal to the given value, which may be a fraction

```
> floor(5)
5
> floor(4.9)
4
```

min takes one or more numbers and returns the smallest number from the set

max takes one or more numbers and returns the greatest number from the set

Log returns the logarithm of a given number in a given base

```
> log(50, 10)
1.6989700043360185
> log(16, 2)
4
```

parseint parses the given string as a representation of an integer in the specified base and returns the resulting number

```
> parseint("100", 10)
100

> parseint("FF", 16)
255

> parseint("-10", 16)
-16

> parseint("1011111011101111", 2)
48879

> parseint("aA", 62)
656

> parseint("12", 2)

Error: Invalid function argument

Invalid value for "number" parameter:
cannot parse "12" as a base 2 integer.
```

pow calculates an exponent, by raising its first argument to the power of the second argument.

```
> pow(3, 2)
9
> pow(4, 0)
1
```

signum determines the sign of a number, returning a number between -1 and 1 to represent the sign

```
> signum(-13)
-1
> signum(0)
0
> signum(344)
1
```

String functions

chomp removes newline characters at the end of a string.

```
> chomp("hello\n")
hello
> chomp("hello\r\n")
hello
> chomp("hello\n\n")
hello
```

format produces a string by formatting a number of other values according to a specification string

```
> format("Hello, %s!", "Ander")
Hello, Ander!
> format("There are %d lights", 4)
There are 4 lights
```

formatlist produces a list of strings by formatting a number of other values according to a specification string

```
> formatlist("Hello, %s!", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Hello, Valentina!",
  "Hello, Ander!",
  "Hello, Olivia!",
  "Hello, Sam!",
]
> formatlist("%s, %s!", "Salutations", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Salutations, Valentina!",
  "Salutations, Ander!",
  "Salutations, Olivia!",
  "Salutations, Sam!",
]
```

indent adds a given number of spaces to the beginnings of all but the first line in a given multi-line string

```
> "  items: ${indent(2, "[\n  foo,\n  bar,\n]\n")}"
  items: [
    foo,
    bar,
  ]
```

join produces a string by concatenating together all elements of a given list of strings with the given delimiter

```
> join(", ", ["foo", "bar", "baz"])
foo, bar, baz
> join(", ", ["foo"])
foo
```

lower converts all cased letters in the given string to lowercase.

```
> lower("HELLO")
hello
```

regex applies a regular expression to a string and returns the matching substrings

```
> regex("^(?:(?P<scheme>[^:/?#]+):)?(?://(?P<authority>[^/?#]*))?", "https://terraform.io/docs/")
{
  "authority" = "terraform.io"
  "scheme" = "https"
}
```

regexall applies a regular expression to a string and returns a list of all matches

replace searches a given string for another given substring, and replaces each occurrence with a given replacement string

```
> replace("hello world", "/w.*d/", "everybody")
hello everybody
```

split produces a list by dividing a given string at all occurrences of a given separator.

```
> split(",", "foo,bar,baz")
[
  "foo",
  "bar",
  "baz",
]
```

strrev reverses the characters in a string

```
> strrev("hello")
olleh
```

substr extracts a substring from a given string by offset and length

```
> substr("hello world", 1, 4)
ello
```

title converts the first letter of each word in the given string to uppercase

```
> title("hello world")
Hello World
```

trim removes the specified characters from the start and end of the given string

```
> trim("?!hello?!", "!?")
hello
```

trimprefix removes the specified prefix from the start of the given string. If the string does not start with the prefix, the string is returned unchanged

```
> trimprefix("helloworld", "hello")
world
```

trimsuffix removes the specified suffix from the end of the given string

```
> trimsuffix("helloworld", "world")
hello
```

trimspace removes all types of whitespace from both the start and the end of a string

```
> trimspace("  hello\n\n")
hello
```

upper converts all cased letters in the given string to uppercase

```
> upper("hello")
HELLO
```

Collection functions

compact takes a list of strings and returns a new list with any empty string elements removed

```
> compact(["a", "", "b", "c"])
[
  "a",
  "b",
  "c",
]
```

concat takes two or more lists and combines them into a single list

```
> concat(["a", ""], ["b", "c"])
[
  "a",
  "",
  "b",
  "c",
]
```

element retrieves a single element from a list

```
> element(["a", "b", "c"], 3)
a
```

Index finds the element index for a given value in a list

```
> index(["a", "b", "c"], "b")
1
```

flatten takes a list and replaces any elements that are lists with a flattened sequence of the list contents

```
> flatten([[["a", "b"], []], ["c"]])
["a", "b", "c"]
```

keys takes a map and returns a list containing the keys from that map

```
> keys({a=1, c=2, d=3})
[
  "a",
  "c",
  "d",
]
```

contains determines whether a given list or set contains a given single value as one of its elements

```
> contains(["a", "b", "c"], "a")
true
> contains(["a", "b", "c"], "d")
false
```

distinct takes a list and returns a new list with any duplicate elements removed

```
> distinct(["a", "b", "a", "c", "d", "b"])
[
  "a",
  "b",
  "c",
  "d",
]
```

length determines the length of a given list, map, or string

```
> length([])
0
> length(["a", "b"])
2
> length({"a" = "b"})
1
> length("hello")
5
```

Lookup retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

```
> lookup({a="ay", b="bee"}, "a", "what?")
ay
> lookup({a="ay", b="bee"}, "c", "what?")
what?
```

merge takes an arbitrary number of maps or objects, and returns a single map or object that contains a merged set of elements from all arguments

```
> merge({a="b", c="d"}, {e="f", c="z"})
{
  "a" = "b"
  "c" = "z"
  "e" = "f"
}
```

one takes a list, set, or tuple value with either zero or one elements. If the collection is empty, one returns null. Otherwise, one returns the first element. If there are two or more elements then one will return an error

```
> one([])
null
> one(["hello"])
"hello"
> one(["hello", "goodbye"])
Error: Invalid function argument
```

Matchkeys constructs a new list by taking a subset of elements from one list whose indexes match the corresponding indexes of values in another list

```
> matchkeys(["i-123", "i-abc", "i-def"], ["us-west", "us-east", "us-east"], ["us-east"])
[
  "i-abc",
  "i-def",
]
```

range generates a list of numbers using a start value, a limit value, and a step value

```
> range(3)
[
  0,
  1,
  2,
]
```

reserve takes a sequence and produces a new sequence of the same length with all of the same elements as the given sequence but in reverse order

```
> reverse([1, 2, 3])
[
  3,
  2,
  1,
]
```

alltrue returns true if all elements in a given collection are true or "true". It also returns true if the collection is empty.

```
> alltrue(["true", true])
true
> alltrue([true, false])
false
```

anytrue returns true if any element in a given collection is true or "true". It also returns false if the collection is empty

```
> anytrue(["true"])
true
> anytrue([true])
true
> anytrue([true, false])
true
> anytrue([])
false
```

setintersection function takes multiple sets and produces a single set containing only the elements that all of the given sets have in common. In other words, it computes the intersection of the sets

```
> setintersection(["a", "b"], ["b", "c"], ["b", "d"])
[
  "b",
]
```

setproduct function finds all of the possible combinations of elements from all of the given sets by computing the Cartesian product.

```
> setproduct(["development", "staging", "production"], ["app1", "app2"])
[
  [
    "development",
    "app1",
  ],
  [
    "development",
    "app2",
  ],
  ...
```

chunklist splits a single list into fixed-size chunks, returning a list of lists

```
> chunklist(["a", "b", "c", "d", "e"], 2)
[
  [
    "a",
    "b",
  ],
  ...
```

coalesce takes any number of arguments and returns the first one that isn't null or an empty string

```
> coalesce("a", "b")
a
> coalesce("", "b")
b
> coalesce(1,2)
1
```

coalescelist takes any number of list arguments and returns the first one that isn't empty

```
> coalescelist(["a", "b"], ["c", "d"])
[
  "a",
  "b",
]
```

Setsubtract function returns a new set containing the elements from the first set that are not present in the second set. In other words, it computes the relative complement of the first set in the second set

```
> setsubtract(["a", "b", "c"], ["a", "c"])
[
  "b",
]
```

setunion function takes multiple sets and produces a single set containing the elements from all of the given sets. In other words, it computes the union of the sets

```
> setunion(["a", "b"], ["b", "c"], ["d"])
[
  "d",
  "b",
  "c",
  "a",
]
```

sum takes a list or set of numbers and returns the sum of those numbers

```
> sum([10, 13, 6, 4.5])
33.5
```

transpose takes a map of lists of strings and swaps the keys and values to produce a new map of lists of strings

```
> transpose({"a" = ["1", "2"], "b" = ["2", "3"]})
{
  "1" = [
    "a",
  ],
  "2" = [
    "a",
    "b",
  ],
  ...
```

slice extracts some consecutive elements from within a list

```
> slice(["a", "b", "c", "d"], 1, 3)
[
  "b",
  "c",
]
```

values takes a map and returns a list containing the values of the elements in that map

```
> values({a=3, c=2, d=1})
[
  3,
  2,
  1,
]
```

sort takes a list of strings and returns a new list with those strings sorted lexicographically

```
> sort(["e", "d", "a", "x"])
[
  "a",
  "d",
  "e",
  "x",
]
```

zipmap constructs a map from a list of keys and a corresponding list of values

```
> zipmap(["a", "b"], [1, 2])
{
  "a" = 1,
  "b" = 2,
}
```

Encoding and decoding functions

Functions that will **encode** and **decode** for various formats

```
> base64encode("Hello World")
SGVsbG8gV29ybGQ=
```

```
> base64decode("SGVsbG8gV29ybGQ=")
Hello World
```

- base64encode
- jsonencode
- textencodebase64
- Yamlencode
- base64gzip
- urlencode

- base64decode
- csvdecode
- jsondecode
- textdecodebase64
- yamldecode

```
> urlencode("Hello World")
Hello%20World
> urlencode("@")
%E2%98%83
> "http://example.com/search?q=${urlencode("terraform urlencode")}"
http://example.com/search?q=terraform%20urlencode
```

Filesystem functions

abspath takes a string containing a filesystem path and converts it to an absolute path. That is, if the path is not absolute, it will be joined with the current working directory

```
> abspath(path.root)
/home/user/some/terraform/root
```

dirname takes a string containing a filesystem path and removes the last portion from it.

```
> dirname("foo/bar/baz.txt")
foo/bar
```

pathexpand takes a filesystem path that might begin with a ~ segment, and if so it replaces that segment with the current user's home directory path

```
> pathexpand("~/.ssh/id_rsa")
/home/steve/.ssh/id_rsa
> pathexpand("/etc/resolv.conf")
/etc/resolv.conf
```

basename takes a string containing a filesystem path and removes all except the last portion from it

```
> basename("foo/bar/baz.txt")
baz.txt
```

file reads the contents of a file at the given path and returns them as a string

```
> file("${path.module}/hello.txt")
Hello World
```

```
> fileexists("${path.module}/hello.txt")
true
```

fileexists determines whether a file exists at a given path

fileset enumerates a set of regular file names given a path and pattern

```
> fileset(path.module, "files/*.txt")
[
  "files/hello.txt",
  "files/world.txt",
]
```

filebase64 reads the contents of a file at the given path and returns them as a base64-encoded string

```
> filebase64("${path.module}/hello.txt")
SGVsbG8gV29ybGQ=
```

templatefile reads the file at the given path and renders its content as a template using a supplied set of template variables

```
%{ for addr in ip_addrs ~}
backend ${addr}:${port}
%{ endfor ~}
```

```
> templatefile("${path.module}/backends.tpl", { port = 8080, ip_addrs = ["10.0.0.1", "10.0.0.2"] })
backend 10.0.0.1:8080
backend 10.0.0.2:8080
```

Date and time functions

formatdate converts a timestamp into a different time format

```
> formatdate("DD MMM YYYY hh:mm ZZZ", "2018-01-02T23:12:01Z")
02 Jan 2018 23:12 UTC
> formatdate("EEEE, DD-MMM-YY hh:mm:ss ZZZ", "2018-01-02T23:12:01Z")
Tuesday, 02-Jan-18 23:12:01 UTC
> formatdate("EEE, DD MMM YYYY hh:mm:ss ZZZ", "2018-01-02T23:12:01-08:00")
Tue, 02 Jan 2018 23:12:01 -0800
> formatdate("MMM DD, YYYY", "2018-01-02T23:12:01Z")
Jan 02, 2018
> formatdate("HH:mmaa", "2018-01-02T23:12:01Z")
11:12pm
```

timeadd adds a duration to a timestamp, returning a new timestamp

```
> timeadd("2017-11-22T00:00:00Z", "10m")
2017-11-22T00:10:00Z
```

timestamp returns a UTC timestamp string in RFC 3339 format

```
> timestamp()
2018-05-13T07:44:12Z
```

Hash and crypto functions

Generate Hashes and cryptographic strings.

```
> bcrypt("hello world")
$2a$10$D5grTTzcsqyvAeIAnY/mYOIqliCoG7eAMX0/oFcuD.iErkksEbcA
```

- base64sha256
- base64sha512
- **bcrypt**
- filebase64sha256
- filebase64sha512
- filemd5
- filesha1
- filesha256
- filesha512
- **md5**
- rsadecrypt
- **sha1**
- sha256
- sha512
- **uuid**
- uuidv5

Network functions

cidrhost calculates a full host IP address for a given host number within a given IP network address prefix

```
> cidrhost("10.12.127.0/20", 16)
10.12.112.16
> cidrhost("10.12.127.0/20", 268)
10.12.113.12
> cidrhost("fd00:fd12:3456:7890:00a2::/72", 34)
fd00:fd12:3456:7890::22
```

cidrnetmask converts an IPv4 address prefix given in CIDR notation into a subnet mask address

```
> cidrnetmask("172.16.0.0/12")
255.240.0.0
```

cidrsubnet calculates a subnet address within given IP network address prefix

```
> cidrsubnet("172.16.0.0/12", 4, 2)
172.18.0.0/16
> cidrsubnet("10.1.2.0/24", 4, 15)
10.1.2.240/28
> cidrsubnet("fd00:fd12:3456:7890::/56", 16, 162)
fd00:fd12:3456:7800:a200::/72
```

cidrsubnets calculates a sequence of consecutive IP address ranges within a particular CIDR prefix.

```
> cidrsubnets("10.1.0.0/16", 4, 4, 8, 4)
[
  "10.1.0.0/20",
  "10.1.16.0/20",
  "10.1.32.0/24",
  "10.1.48.0/20",
]
```

Type conversion functions

can evaluates the given expression and returns a boolean value indicating whether the expression produced a result without any errors

```
> can(local.foo.bar)
true
```

defaults a specialized function intended for use with input variables whose type constraints are object types or collections of object types that include optional attributes

nonsenstive takes a sensitive value and returns a copy of that value with the sensitive marking removed, thereby exposing the sensitive value

```
output "sensitive_example_hash" {
  value = nonsensitive(sha256(var.sensitive_example))
}
```

sensitive takes any value and returns a copy of it marked so that Terraform will treat it as sensitive, with the same meaning and behavior as for sensitive input variables.

```
locals {
  sensitive_content = sensitive(file("${path.module}/sensitive.txt"))
}
```

tobool converts its argument to a boolean value

```
tobool("true")
true
```

tolist converts its argument to a list value

```
> tolist(["a", "b", 3])
[
  "a",
  "b",
  "3",
]
```

tomap converts its argument to a map value

```
> tomap({"a" = 1, "b" = 2})
{
  "a" = 1
  "b" = 2
}
```

tonumber converts its argument to a number value

```
> tonumber("1")
1
```

toset converts its argument to a set value.

```
> toset(["a", "b", "c"])
[
  "a",
  "b",
  "c",
]
```

tostring converts its argument to a set value

```
> tostring(true)
"true"
```

try evaluates all of its argument expressions in turn and returns the result of the first one that does not produce any errors

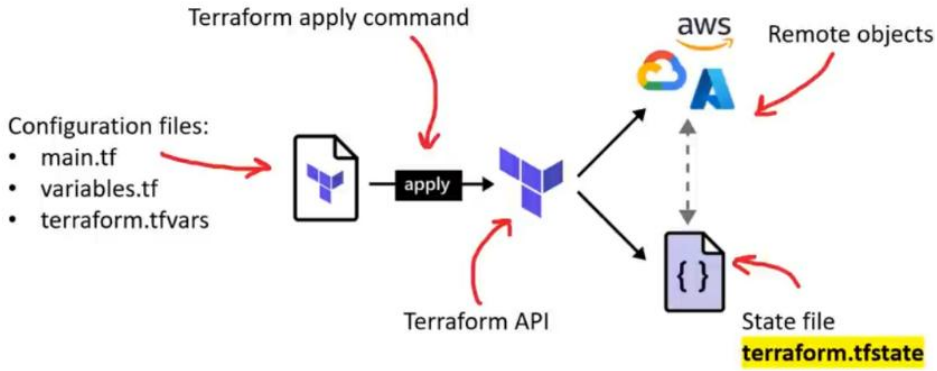
```
locals {
  example = try(
    [tostring(var.example)],
    tolist(var.example),
  )
}
```

Terraform State

- Terraform state is a critical component that tracks the relationship between your configuration and the real-world infrastructure it manages.
- It serves as Terraform's "source of truth" about your deployed resources.
- **What is Terraform State?**
  - The state file (`terraform.tfstate`) is a JSON document that:
    - Maps resources in your configuration to real infrastructure
    - Stores resource attributes and dependencies
    - Tracks metadata about your infrastructure
- **Why State is Essential**
  - Performance
    - Stores resource attributes so Terraform doesn't need to refresh every attribute from providers
  - Dependency Tracking
    - Records relationships between resources
  - Drift Detection
    - Compares actual infrastructure with configuration
  - Collaboration
    - Enables team workflows when stored remotely

What is State?
A particular condition of cloud resources at a specific time.
eg. We expect there to be a Virtual Machine running CentOS on AWS with a compute type of t2.micro.

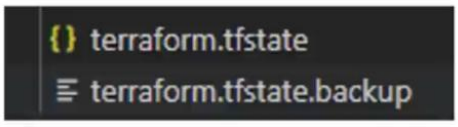
How does Terraform preserve state?
When you provision infrastructure via Terraform it will create a state file named **terraform.tfstate**
This **state file is a JSON data structure** with a one-to-one mapping from **resource instances** to **remote objects**



terraform state backup

All terraform state subcommands that *modify state* will write a backup file.
Read only commands will not modify state eg. list, show

Terraform will take the current state and store it in a file called **terraform.tfstate.backup**



Backups cannot be disabled. This is by design to enforce best-practice for recovery

To get rid of the backup file you need to manually delete the files

Dependency lock file vs State lock file

Dependency Lock File (.terraform.lock.hcl)

- **Purpose:**
  - Locks provider/plugin versions to ensure consistent deployments
  - Prevents automatic upgrades that might break your configuration
- **Location:**
  - `.terraform.lock.hcl` in your Terraform directory
- **Key Features:**
  - Generated by `terraform init`
  - Lists exact provider versions and checksums
  - Committed to version control (Git)
  - Affects all workspaces using the configuration
- **Example Content:**

```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "4.55.0"
  constraints = "~> 4.0"
  hashes = [
    "h1:vSVjfh4GIrca2Z3YPjWMaac5hOEBc1U3xOOwFmo7HZc=",
  ]
}
```

- **When It's Used:**
  - During terraform init to download correct provider versions
  - Prevents version mismatches across teams
- **How to Update:**

```
terraform init -upgrade  # Updates lock file to newest allowed versions
```

State Lock File (Advisory Lock)

- **Purpose:**
  - Prevents concurrent operations that might corrupt state
  - Ensures only one Terraform operation runs at a time
- **Location:**
  - Local: Temporary file (`.terraform.tfstate.lock.info`)
  - Remote: Backend-specific (e.g., DynamoDB table for AWS)
- **Key Features:**
  - Created during terraform apply/plan/destroy
  - Automatically released when operation completes
  - Not committed to version control
  - Workspace-specific
- **Example Mechanism:**

```
# AWS S3 backend with DynamoDB locking
terraform {
  backend "s3" {
    bucket         = "my-state-bucket"
    key            = "prod/terraform.tfstate"
    dynamodb_table = "terraform-locks"  # Locking table
  }
}
```

Or the new version:

```
terraform {
  backend "s3" {
    bucket       = "mybucket"
    key          = "path/to/my/key"
    region       = "us-east-1"
    use_lockfile = true
  }
}
```

- **When It's Used:**
    - When running operations that modify state
    - Prevents two team members from running apply simultaneously
- **How to Release:**

```
terraform force-unlock LOCK_ID  # Only if stuck
```

Backends

- Backends in Terraform define where and how state is stored and how operations are executed.
- They are crucial for team collaboration, security, and workflow optimization.
- **A backend determines:**
  - State storage location (local file, S3, Terraform Cloud, etc.)
  - State locking (to prevent concurrent operations)
  - Operation execution (local vs. remote)

Backend Types

- **Local Backend (Default)**

The local backend:
- stores state on the local filesystem
- locks that state using system APIs
- performs operations locally

By default, you are using the backend state when you have no specified backend

```
terraform {
}
```

You specific the backend with argument **local**, and you can change the path to the local file and **working_directory**

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

You can set a backend to reference another state file so you can read its outputted values.

This is way of **cross-referencing stacks**

```
data "terraform_remote_state" "networking" {
  backend = "local"

  config = {
    path = "${path.module}/networking/terraform.tfstate"
  }
}
```

- يبقى الLocal Backend بيتميز انه بيديني الحاجتين سوى ، بيعمل تخزين للState File ، واقدر من خلاله اعمل Terraform Operations زى `terraform apply` مثلا.

- **Remote Backends**
  - *Standard Remote (State Only)*
    - Stores state remotely
    - Operations run locally
    - The backup of the state file will still reside on your local machine.

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "prod/network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

- يبقى اول نوع من أنواع الRemote Backend هو الStandard Remote ودة بيقدر يخزن الState File لكن مش بيقدر يعمل الTerraform Operations وبالتالي بعملها عندى Local على الجهاز بتاعى.

- *Enhanced Remote (Terraform Cloud/Enterprise)*
  - Uses terraform platform (Cloud/Enterprise)
  - Stores state remotely
  - Can execute operations remotely (optional)

```
terraform {
  backend "remote" {
    organization = "my-org"
    workspaces {
      name = "prod-network"
    }
  }
}
```

When using Terraform Cloud as a remote backend state you should instead use the cloud block to configure its usage.

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      prefix = "my-app-"
    }
  }
}
```

```
terraform {
  cloud {
    organization = "company"
    hostname = "app.terraform.io"

    workspaces {
      tags = ["app"]
    }
  }
}
```

You can still use backend remote for terraform cloud but its recommended to use the cloud backend block.

- النوع التانى من الRemote Backend هو الTerraform Platform سواء كان Entrprise او Cloud ودة ميزته بقى انى اقدر اعمل فيه الحاجتين (State File – Terrform Operations).

Backend Initialization

The **-backend-config** flag for **terraform init** can be used for partial backend configuration

In situations where the backend settings are dynamic or sensitive and so cannot be statically specified in the configuration file.

main.tf

```
terraform {
  required_version = "~> 0.12.0"

  backend "remote" {}
}
```

backend.hcl

```
workspaces { name = "workspace" }
hostname     = "app.terraform.io"
organization = "company"
```

```
terraform init -backend-config=backend.hcl
```

terraform-remote-state

terraform_remote_state data source **retrieves the root module output values from another Terraform configuration**, using the latest state snapshot from the remote backend.

Remote Backend

```
data "terraform_remote_state" "vpc" {
  backend = "remote"

  config = {
    organization = "hashicorp"
    workspaces = {
      name = "vpc-prod"
    }
  }
}
resource "aws_instance" "foo" {
  subnet_id = data.terraform_remote_state.vpc.outputs.subnet_id
}
```

Local Backend

```
data "terraform_remote_state" "vpc" {
  backend = "local"

  config = {
    path = "..."
  }
}

resource "aws_instance" "my_server" {
  #
  subnet_id = data.terraform_remote_state.vpc.outputs.subnet_id
}
```

- Only the root-level output values from the remote state snapshot are exposed
- Resource data and output values from nested modules are not accessible
- To make a nested module output value accessible as a root module output value, you must **explicitly configure a passthrough in the root module**

```
module "app" {
  source = "..."
}

output "app_value" {
  value = module.app.example
}
```

terraform_remote_state only exposes output values, its user must have access to the entire state snapshot, which often includes some sensitive information.

It recommend explicitly publishing data for external consumption to a separate location instead of accessing it via remote state

Use **a data source** and reference resources whenever you can:

```
data "aws_s3_bucket" "selected" {
  bucket = "bucket.test.com"
}

data "aws_route53_zone" "test_zone" {
  name = "test.com."
}

resource "aws_route53_record" "example" {
  zone_id  = "${data.aws_route53_zone.test_zone.id}"
  name     = "bucket"
  type     = "A"

  alias {
    name    = "${data.aws_s3_bucket.selected.website_domain}"
    zone_id = "${data.aws_s3_bucket.selected.hosted_zone_id}"
  }
}
```

Why do we need a remote backend?

- When working with Terraform locally, the state files are created in the project's root directory. This is known as the local backend and is the default.
- These files cannot be a part of the Git repository as every member may execute their version of Terraform config and end up modifying the state files.
- Thus, there is a high chance of corrupting the state file or at least creating inconsistencies in the state files – which is equally dangerous.
- Managing the state files in a remote Git repository is also discouraged as the state files may contain sensitive information like credentials, etc.

State locking

- State locking is a critical mechanism that prevents concurrent operations on the same Terraform state, ensuring consistency and preventing corruption.
- **What is State Locking?**
  - An advisory lock mechanism that blocks multiple simultaneous operations.
  - Prevents two team members from running `terraform apply` at the same time.
  - Ensures state files aren't corrupted by parallel writes.
- **How Locking Works**
  - *Lock Lifecycle*
    - Acquisition: Lock is obtained when an operation starts (`plan`/`apply`/`destroy`).
    - Holding: Lock persists during the operation.
    - Release: Lock is released when operation completes (success or failure).

- يبقى اللى بيحصل لما باجى اعمل commands زى terraform plan او terraform apply ، تيرافورم بيحاول يـ acquire a lock on the state file ، فلو لقى ان مفيش Locking موجود بيعمل lock ويبدء يشتغل ، لو مفيش بيستنى لحد ما يحصل release وبعدين يعمل lock ويشتغل.

    - *Lock Contents*
        - Typical lock includes:
            - Lock ID (unique identifier)
            - Who created the lock (user/machine)
            - Timestamp
            - Operation type (apply, plan, etc.)
- **Locking Backends**
    - *Local Backend*
        - Uses local filesystem locks (`.terraform.tfstate.lock.info`)
        - Limitation: Doesn't work for teams
        - **Example:**

```
$ cat terraform.tfstate.lock.info
{
  "ID":"12345-abcde",
  "Operation":"apply",
  "Info":"",
  "Who":"user@machine",
  "Version":"1.3.7",
  "Created":"2023-05-20T12:00:00Z",
  "Path":"terraform.tfstate"
}
```

    - *Remote Backends with Locking*

Backend	Locking Mechanism
AWS S3	DynamoDB table or S3 (new versions)
Terraform Cloud	Built-in locking
AzureRM	Blob storage lease
Google Cloud Storage	Cloud Storage lock
HashiCorp Consul	Consul sessions

Force-Unlock

- The `force-unlock` command is a last-resort tool for manually releasing a stuck Terraform state lock when normal operations fail to release it.
- **When to Use Force-Unlock**
    - Terraform process crashed mid-operation
    - Network issues prevented lock release
    - Legitimate stale lock (operator left company)
- **Never Use When**
    - Another team member is actively running Terraform
    - You're unsure why the lock exists
    - As a routine troubleshooting step
- **Basic Command Syntax**

```
terraform force-unlock LOCK_ID
```

```
terraform force-unlock 1941a539b-ff25-76ef-92d-547ab37b24d
```

Step-by-Step Process

- **Identify the Lock**
  - Try a normal operation first:

```
terraform plan
```

    - The error message will show the lock ID and owner

  - For AWS S3+DynamoDB backends:

```
aws dynamodb scan --table-name YOUR_LOCK_TABLE
```

- **Verify Safety**
  - Check who holds the lock (from error message)
  - Confirm no active operations are running
  - Notify your team about the forced unlock
- **Execute Unlock**

```
terraform force-unlock a1b2c3d4-5678-90ef-ghij-klmnopqrstuv
```

- **Verify Release**

```
terraform plan  # Should now work
```

Protecting sensitive data

Terraform State file **can contain sensitive data** eg. long-lived AWS Credentials and is a possible **attack vector** for malicious actors.

Local State
When using local backend, state is stored in plain-text JSON files.
- You need to be careful you don't share this state file with anyone
- You need to be careful you don't commit this file to your git repository

Remote State with Terraform Cloud
When using the Terraform Cloud remote backend:
- That state file is held in memory and is not persisted to disk
- The state file is encrypted-at-rest
- The state file is encrypted-in-transit
- With Terraform Enterprise you have detailed audit logging for tamper evidence

Remote State with Third-Party Storage
You can store state with various third-party backends.
You need to carefully review your backends capabilities to determine if will meet your security and compliance requirements.
Some backends are not by default as secure as they could be:
- eg. With AWS S3 you have to ensure encryption and versioning is turned on, you need to create a custom trail for data events

Terraform ignore file

When executing a **remote** plan or apply in a CLI-driven run,
an archive of your configuration directory is uploaded to Terraform Cloud.

You can define paths to ignore from upload via a **.terraformignore** file at the root of your configuration directory.

If this file is not present, the archive will exclude the following **by default**:
- .git/ directories
- .terraform/ directories (exclusive of .terraform/modules)

.terraformignore works just like a .gitignore with the only difference is that you cannot have multiple .terraformignore files in subdirectories. Only the the file in the root directory will be read

Provisioners

- Provisioners in Terraform are used to execute scripts or commands on local or remote machines after resource creation.
- They help with:
    - Bootstrapping (e.g., installing software on a new VM).
    - Configuring resources (e.g., running database migrations).
    - Handling edge cases where Terraform's declarative model isn't enough.

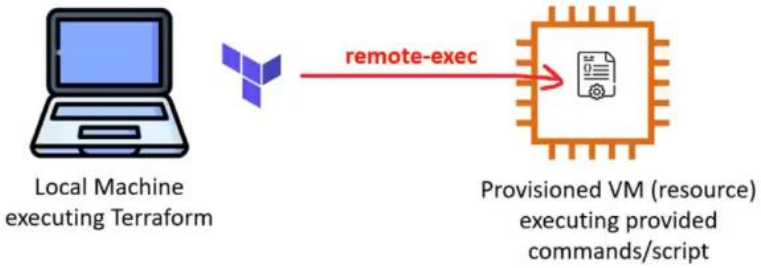
local-exec (Run Commands Locally)

- Executes a command on the machine running Terraform (e.g., your laptop or CI/CD runner) after a resource is provisioned.
- **Example:**

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo 'Instance created at ${self.public_ip}' > ip.txt"
  }
}
```

- **Use Cases:**
    - Notifying external systems (e.g., Slack, monitoring tools).
    - Generating local configuration files.

remote-exec (Run Commands on the Resource)



- Executes commands directly on the newly created resource (e.g., via SSH or WinRM).
- `remote-exec` requires a `connection` block to define how Terraform accesses the resource.
- **Example (SSH):**

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  # Ensure SSH is ready
  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
```

```
      "sudo apt-get install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

- **Use Cases:**
  - Installing software (e.g., Docker, Nginx).
  - Configuring services (e.g., editing config files).
- **Modes:**
  - `inline` Mode (Default)
    - Executes a list of commands sequentially on the remote machine.
    - **Example (SSH)**

```
provisioner "remote-exec" {
  inline = [
    "sudo apt-get update",
    "sudo apt-get install -y nginx",
    "sudo systemctl start nginx"
  ]
}
```

  - `script` Mode
    - Uploads a local script and executes it remotely.
    - **Example**

```
provisioner "remote-exec" {
  script = "./setup-server.sh"  # Local script path
}
```

  - `scripts` Mode
    - Uploads and runs multiple scripts in order.
    - **Example**

```
provisioner "remote-exec" {
  scripts = [
    "./install-dependencies.sh",
    "./deploy-app.sh"
  ]
}
```

file (Copy Files to the Resource)

file provisioner is used to copy files or directories from our local machine to the newly created resource

Source – the local file we want to upload to the remote machine

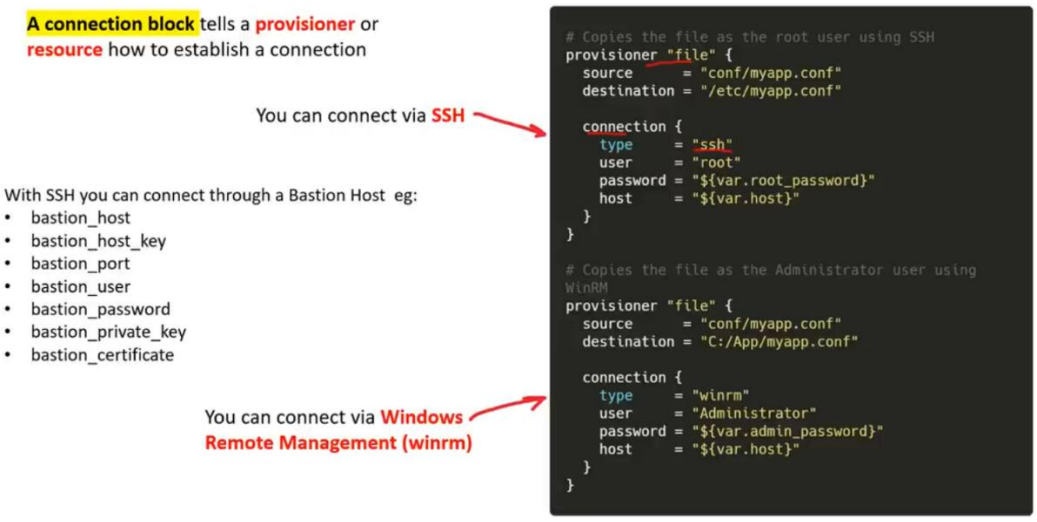
Content – a file or a folder

Destination – where you want to upload the file on the remote machine

You may require a connection block within the provisioner for authentication

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }

  # Copies all files and folders in apps/app1 to
D:/IIS/webapp1
  provisioner "file" {
    source      = "apps/app1/"
    destination = "D:/IIS/webapp1"
  }
}
```

Connections



Null_resource

- The `null_resource` is a special Terraform resource that does not create any actual infrastructure.
- Instead, it acts as a placeholder to:
  - Trigger actions (like running local/remote scripts).
  - Manage dependencies between resources.
  - Simulate a resource when no real cloud provider resource is needed.
- It is part of the Terraform "null" provider, which is included by default (no need to declare it in `required_providers`).
- **Use cases:**
  - Running Provisioners Without a Real Resource
    - Execute `local-exec` or `remote-exec` scripts without tying them to a cloud resource (e.g., AWS EC2).
  - Triggering Actions Based on Changes
    - Re-run scripts when dependencies update (e.g., a file or another resource changes).
  - Workarounds for Terraform Limitations
    - When you need a "dummy" resource to control execution order.
- **Key Features**
  - *Triggers (Forced Re-Execution)*
    - The triggers argument forces the `null_resource` to re-run its provisioners when the referenced values change.
    - **Example:**

```
resource "null_resource" "deploy" {
  triggers = {
    # Re-run if the app code changes
    app_version = filemd5("${path.module}/app.zip")
  }

  provisioner "local-exec" {
    command = "aws s3 cp app.zip s3://my-bucket/"
  }
}
```

    - If `app.zip` is modified, Terraform will re-run the `local-exec` command.

- *Provisioners (local-exec / remote-exec)*
  - You can attach provisioners to a `null_resource` just like any other resource.
  - **Example with local-exec:**

```
resource "null_resource" "notify_slack" {
  provisioner "local-exec" {
    command = "curl -X POST -H 'Content-type: application/json' --data '{\"text\":\"Deployment started\"}' $SLACK_WEBHOOK"
  }
}
```

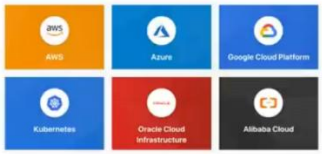
  - **Example with remote-exec (SSH):**

```
resource "null_resource" "configure_server" {
  triggers = {
    instance_id = aws_instance.web.id
  }

  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
    host        = aws_instance.web.public_ip
  }

  provisioner "remote-exec" {
    inline = ["sudo systemctl restart nginx"]
  }
}
```

- *Dependency Management*
  - Since Terraform creates resources in parallel, `null_resource` can enforce explicit dependencies.
  - **Example:**

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}

# Wait for the EC2 instance to be ready before running the script
resource "null_resource" "setup" {
  depends_on = [aws_instance.web]  # Explicit dependency

  provisioner "local-exec" {
    command = "./init-backend.sh ${aws_instance.web.private_ip}"
  }
}
```

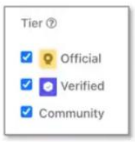
Providers



Providers are Terraform Plugins that allow you to interactive with:

- Cloud Service Providers (CSPs) eg. **AWS, Azure, GCP**
- Software as a Service (SaaS) Providers eg. **Github, Angolia, Stripe**
- Other APIs eg. **Kubernetes, Postgres**

Providers are required for your Terraform Configuration file to work.



Providers come in three tiers:
Official — Published by the company that owns the provider technology or service
Verified — actively maintained, up-to-date and compatible with both Terraform and Provider
Community — published by a community member but no guarantee of maintenance, up-to-date or compatibility

Providers are distributed separately from Terraform and the plugins must be downloaded before use.
terraform init will download the necessary provider plugins listed in a Terraform configuration file.

- **Key Concepts**
  - *Provider Configuration*
    - Configure authentication and default settings:

```
provider "aws" {
  region = "us-east-1"       # Default region
  profile = "my-aws-profile" # AWS CLI profile
}
```

  - *Multiple Provider Instances (Aliasing)*
    - Provider aliasing allows you to:
      - Manage multi-region deployments
      - Work with multiple accounts in the same cloud
      - Support multi-cloud configurations
      - Isolate environments (dev/prod) cleanly

```
# Default provider
provider "aws" {
  region = "us-east-1"
  profile = "prod-profile"
}

# Aliased provider
provider "aws" {
  alias  = "west"
  region = "us-west-2"
  profile = "dev-profile"
}

resource "aws_instance" "east" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
  # Uses default provider (us-east-1)
}

resource "aws_instance" "west" {
  provider      = aws.west  # Explicit alias
  ami           = "ami-789012"
  instance_type = "t2.micro"
}
```

Modules

- Modules are reusable, configurable infrastructure packages that help you standardize and scale your Terraform configurations.
- They encapsulate resources, variables, and outputs into shareable components.
- **What Modules Solve**
    - *Code duplication* - Avoid copying/pasting the same VPC setup across projects
    - *Standardization* - Enforce best practices (e.g., tagging, security)
    - *Abstraction* - Hide complexity behind simple interfaces
- **Core Components**

```
modules/
└── vpc/
    ├── main.tf       # Resource definitions
    ├── variables.tf  # Input variables
    ├── outputs.tf    # Output values
    └── README.md     # Documentation
```

- **Types of Modules**
    - *Local Modules*
        - Reusable within your codebase:

```
module "network" {
  source = "./modules/vpc"  # Local path
  cidr   = "10.0.0.0/16"
}
```

    - *Public Registry Modules*
        - From Terraform Registry:

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "~> 19.0"  # Version constraint
}
```

    - *Private Modules*
        - Hosted in Terraform Cloud/Enterprise:

```
module "internal_app" {
  source  = "app.terraform.io/acme/vpc/aws"
  version = "1.2.0"
}
```

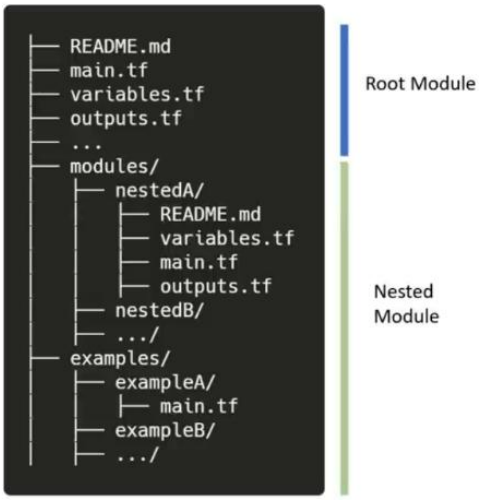
The Standard Module Structure is a file and directory layout recommend for module development

The primary entry point is the **Root Module.**
These are required files in the root directory:

- Main.tf - the entry point file of your module
- Variables.tf – variable that can be passed in
- Outputs.tf – Outputed values
- README – Describes how the module works
- LICENSE – The license under which this module is available

Nested modules are optional must be contained in the **modules/** directory:

- A submodule that contains a README is consider usable by external users
- A submodule that does not contain a README is consider internal use only
- Avoid using relative paths when sourcing module blocks.



- **Creating a Module**
    - Directory structure:

```
modules/
└── s3/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

    - Define resources (`main.tf`):

```
resource "aws_s3_bucket" "this" {
  bucket = var.bucket_name
  acl    = var.acl
  tags   = var.tags
}
```

    - Declare inputs (`variables.tf`):

```
variable "bucket_name" {
  type = string
}
variable "acl" {
  type    = string
  default = "private"
}
variable "tags" {
  type    = map(string)
  default = {}
}
```

    - Expose outputs (`outputs.tf`):

```
output " bucket_regional_domain_name " {
  value = aws_s3_bucket.this.bucket_regional_domain_name
}
```

- **Using Modules**

```
module "logs" {
  source      = "./modules/s3"
  bucket_name = "my-app-logs"
  tags = {
    Environment = "prod"
  }
}

# Reference module output
resource "aws_cloudfront_distribution" "cdn" {
  origin {
    domain_name = module.logs.bucket_regional_domain_name
    # ...
  }
}
```

- **Advanced Features**
    - Count/For Each:

```
module "buckets" {
  for_each    = toset(["assets", "logs", "backups"])
  source      = "./modules/s3"
  bucket_name = "myapp-${each.key}"
}
```

    - Providers:

```
module "cross_account" {
  source    = "./modules/vpc"
  providers = {
    aws = aws.replica_region
  }
}
```

Drift Detection and Management

- Drift occurs when the actual state of your infrastructure differs from what Terraform has recorded in its state file.
- This is a critical concept for maintaining infrastructure reliability.
- **Drift happens when:**
    - Someone manually changes cloud resources
    - External processes modify infrastructure
    - APIs/SDKs make changes outside Terraform
    - Terraform fails to fully apply changes
    - **Example:** A team member manually changes an EC2 instance type from t3.medium to m5.large in AWS Console, making it diverge from Terraform's configuration.
- **How Terraform Detects Drift**
    - *Automatic Detection*

```
terraform plan  # Compares state with real infrastructure
```

        - Output shows:

```
# aws_instance.web will be updated in-place
~ resource "aws_instance" "web" {
    ~ instance_type = "t3.medium" -> "m5.large" # (forces replacement)
      tags          = {}
      # (...)
  }
```

    - *Manual Detection Methods*
        - Refresh State:

```
terraform refresh  # Updates state file with real-world data
terraform plan     # Then check differences
```

        - Cloud-Specific Tools:
            - AWS: AWS Config Rules
            - Azure: Azure Policy
            - GCP: Security Health Analytics
- **Types of Drift**

Drift Type	Example	Terraform Response
Configuration Drift	Manual EC2 size change	plan shows update needed
Resource Deletion	S3 bucket deleted via CLI	plan shows recreation required
Metadata Drift	Tags modified externally	Depends on ignore_changes
Provider Drift	AWS changes API behavior	May require provider update

Terraform Drift Resolution Methods

- **Replacing Resources (`-replace`)**
    - *When to Use:*
        - Resource is corrupted (e.g., AWS instance won't boot)
        - Terraform can't auto-detect the issue but you know it needs replacement
    - *Command:*

```
terraform apply -replace="aws_instance.web"
```

- *What Happens:*
  - Terraform destroys and recreates the specific resource
  - Preserves other resources

- **Importing Resources (`import`)**
  - *When to Use:*
    - Someone manually created infrastructure you now want to manage with Terraform
    - Resources exist outside Terraform's state
  - *Step-by-Step Import Process:*
    - Define a placeholder in your configuration:

```
resource "aws_s3_bucket" "legacy" {
  # May start empty and fill it after importing or start with known values
}
```

    - Run import:

```
terraform import aws_s3_bucket.legacy my-existing-bucket-name
```

    - Verify state:

```
terraform state show aws_s3_bucket.legacy
```

    - Update configuration manually to align with imported resource:

```
resource "aws_s3_bucket" "legacy" {
  bucket = "my-existing-bucket-name"
  acl    = "private"  # Add attributes from state inspection
}
```

    - Plan/Apply to ensure consistency:

```
terraform plan  # Should show no changes if config matches
```

  - *What Happens:*
    - Terraform adds the existing resource to state
    - Next `apply` will align configuration with actual resource
  - *Key Limitations*
    - Single-resource imports: No bulk imports
    - Not all resources are importable: Check provider docs

- **Refreshing State (`-refresh-only`)**
  - *When to Use:*
    - Manual changes were intentionally made (e.g., resized RDS instance)
    - You want to accept these changes into Terraform's state
  - *Command:*

```
terraform apply -refresh-only
```

  - *What Happens:*
    - Updates state file to match real infrastructure
    - Doesn't modify actual resources
  - *Example:*

```
# After manually upgrading an RDS instance class:
terraform apply -refresh-only -auto-approve
```

The **terraform refresh** command **reads the current settings** from all managed remote objects **and updates the Terraform state to match.**

The terraform refresh command is an alias for the refresh only and auto approve

```
terraform refresh
```

```
terraform apply -refresh-only -auto-approve
```

Terraform refresh will not modify your real remote objects, but it will modify the Terraform state.

Terraform refresh has been ***deprecated*** and with the refresh-only flag because it was not safe since it did not give you an opportunity to review proposed changes before updating state file

The **–refresh-only** flag for terraform plan or apply allows you to refresh and update your state file without making changes to your remote infrastructure.

Scenario
Imagine you create a terraform script that deploys a Virtual Machine on AWS
You ask an engineer to terminate the server, and instead of updating the terraform script they mistakenly terminate the server via the AWS Console.

You run....

terraform apply

- Terraform will notice that the VM is missing
- Terraform will propose to create a new VM

The State File is correct
Changes the infrastructure to match state file

terraform apply **–refresh-only**

- Terraform will notice that the VM you provisioned is missing.
- With the refresh-only flag that the the missing VM is intentional
- Terraform will propose to delete the VM from the state file

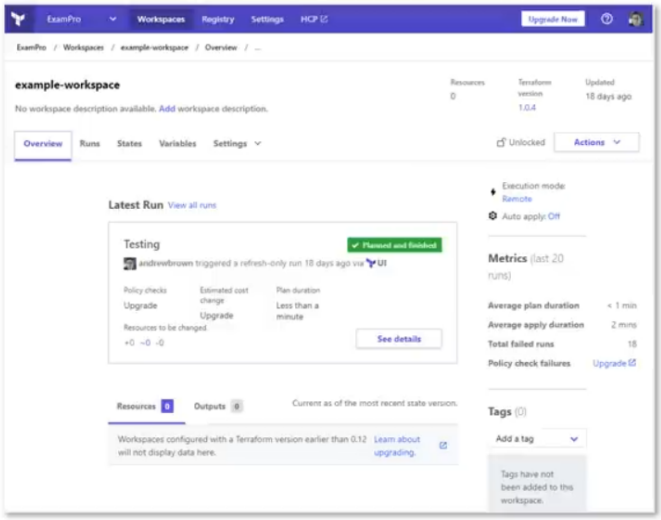
The State File is wrong
Changes the state file to match infrastructure

Key Insights

- Replacement is for "broken" resources
- Importing is for "unmanaged" resources
- Refresh is for "approved manual changes"

Terraform cloud

Terraform Cloud is an application that helps teams use Terraform together.



Terraform Cloud features:

- Manages your state files
- History of previous runs
- History of previous states
- Easy and secure variable injection
- Tagging
- Run Triggers (chaining workspaces)
- Specify any version of terraform per workspace
- Global state sharing
- Commenting on Runs
- Notifications via Webhooks, Email, Slack
- Organization and Workspace Level Permissions
- Policy as Code (via Sentinel Policy Sets)
- MFA,
- Single Sign On (SSO) (at Business tier)
- Cost Estimation (at Teams and Governance Tier)
- Integrations with ServiceNow, Splunk, K8, and custom Run Tasks

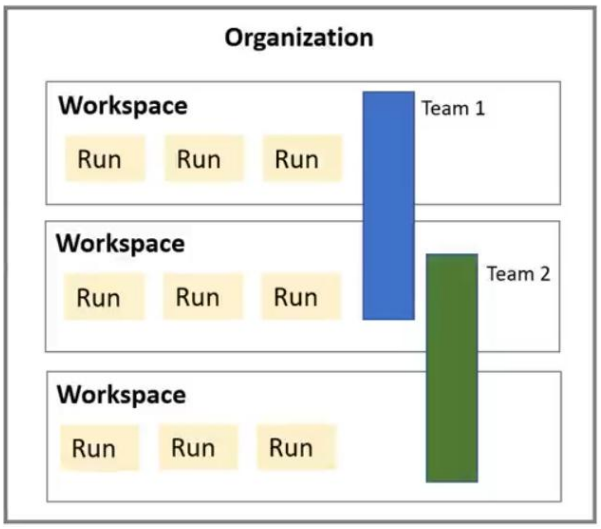
Terraform Cloud is available as a hosted service at app.terraform.io

Organizations
An organization is a collection of workspaces

Workspaces
A workspace belongs to an organization
A workspace represents a unique environment or stack.

Teams
A team is composed of multiple members (users).
A team can be assigned to workspaces

Runs
A run represents a single-run of the terraform run environment that is operating on a execution plan.
Runs can be UI/VCS driven API driven or CLI driven



Terraform cloud run workflows

Terraform Cloud offers **3 types** of Cloud Run Workflows

UI/VCS Driven (User Interface and Version Control System)
Terraform Cloud is integrated with a specific branch in your VCS eg. Github via webhooks.
Whenever pull requests are submitted for the branch speculative plans are generated
Whenever a merge occurs to that branch, than a run is triggered on Terraform Cloud

API-Driven (Application Programming Interface)
Workspaces are not directly associated with a VCS repo, and runs are not driven by webhooks on your VCS provider
A third-party tool or system will trigger runs via upload a configuration file via the Terraform Cloud API
The configuration file is a bash script that is packaged in an archive (.tar.gz). You are pushing a **Configuration Version**

CLI-Driven (Command Line Interface)
Runs are triggered by the user running terraform CLI commands e.g. terraform apply and plan locally on their own machine.

Organization-level permissions

Organization-Level Permissions manage certain resources or settings **across an organization**

- **Manage Policies** - create, edit, and delete the organization's Sentinel policies
- **Manage Policy Overrides** - override soft-mandatory policy checks.
- **Manage Workspaces** - create and administrate all workspaces within the organization
- **Manage VCS Settings** - set of VCS providers and SSH keys available within the organization

Organization Owners

Every organization has organization owner(s).

This is a special role that has every available permission and some actions only available to owners:

- Publish private modules
- Invite users to organization
- Manage team membership
- View all secret teams
- Manage organization permissions
- Manage all organization settings
- Manage organization billing
- Delete organization
- Manage agent

Workspace-level permissions

Workspace-Level Permissions manage resource and settings for a **specific workspace**

General Workspace Permissions

Granular permissions you can apply to a user via **custom workspace permissions**

- Read runs
- Queue plans
- Apply runs
- Lock and unlock workspaces
- Download sentinel mocks
- Read variable
- Read and write variables
- Read state outputs
- Read state versions
- Read and write state versions

Fixed Permission Sets

Premade permissions for quick assignment.

- **Read**
  - Read runs
  - Read variables
  - Read state versions
- **Plan**
  - Queue Plans
  - Read variables
  - Read state versions
- **Write**
  - Apply runs
  - Lock and unlock workspaces
  - Download Sentinel mocks
  - Read and write variables
  - Read and write state versions

Workspace Admins

A workspace admin is a special role that grants the all level of permissions and some workspace-admin-only permissions:

- Read and write workspace settings
- Set or remove workspace permissions of any team

Terraform Workflows

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.

As you team and requirements grow your workflow will evolve.
Lets look at what this workflow will look like for:

- Individual Practitioner (One person team)
- Teams using OSS
- Teams using Terraform Cloud

Individual Practitioner

Write

- You write your Terraform configuration in your editor of choice
- You'll store your terraform code in a VCS e.g. Github
- You repeatedly run terraform plan and validate to find syntax errors.
- Tight feedback loop between editing code and running test commands

Plan

- When the developer is confident with their work in the write step they commit their code to their local repository.
- They may be only using a single branch e.g. main
- Once their commit is written they'll proceed to apply

Apply

- They will run terraform apply and be prompted to review their plan
- After their final review they will approve the changes and await provisioning
- After a successful provision they will push their local commits to their remote repository.

Team

Write

- Each team members writes code locally on their machine in their editor of choice
- A team member will store their code to a branch in their code repository
  - Branches help avoid conflicts while a member is working on their code
  - Branches will allow an opportunity to resolve conflict during a merge into main
- Terraform plan can be used as a quick feedback loop for small teams
- For larger teams a concern over sensitive credentials becomes a concern.
  - A CI/CD process may be implemented so the burden of credentials is abstracted away

Plan

- When a branch is ready to be incorporated on Pull Request an Execution Plan can be generated and displayed within the Pull Request for Review

Apply

- To apply the changes the merge needs to be approved and merged, which will kick of a code build server that will run terraform apply.

- The DevOps team has to setup and maintain their own CI/CD Pipeline
- They have to figure out how to store the state file e.g. Standard Backend remote state
- They are limited in their access controls (they can't be granular about what actions are allowed to performed by certain members e.g. apply, destroy
- They have to figure out a way to safely store and inject secrets into their build server's runtime environment
- Managing multiple environments can make the overhead of the infrastructure increase dramatically.

Team using Terraform Cloud

Write

- A team will use Terraform Cloud as their remote backend.
- Inputs Variables will be stored on Terraform Cloud instead of the local machines.
- Terraform Cloud integrates with your VCS to quickly setup a CI/CD pipeline
- A team member writes code to branch and commits per usual

Plan

- A pull request is create by a team member and Terraform Cloud will generate the speculative plan for review in the VCS. The member can also review and comment on the plan in Terraform Cloud.

Apply

- After the Pull request is merge Terraform Cloud runtime environment will perform a Terraform apply. A team member can Confirm and apply the changes.

Terraform Cloud streamlines a lot of the CI/CD effort, storing and securing sensitive credentials and makes it is easier to go back and audit the history of multiple runs.

Troubleshooting

There are **four types of errors** you can encounter with Terraform:

Language errors
Terraform encounters a syntax error in your configuration for the Terraform or HCL Language

terraform fmt
terraform validate
terraform version

State errors
Your resources state has changed from the expected state in your configuration file

terraform refresh
terraform apply
terraform –replace flag

Core errors
A bug has occurred with the core library

TF_LOG
Open Github Issue

Provider errors
The provider's API has changed or does not work as expected due to emerging edge cases

TF_LOG
Open Github Issue

Easy to Solve | **Harder to Solve**

Debugging

Terraform has detailed logs which can be enabled by setting the **TF_LOG environment** variable to:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- JSON — outputs logs at the TRACE level or higher, and uses a parseable JSON encoding as the formatting.

Logging can be enabled separately:
- TF_LOG_CORE
- TF_LOG_PROVIDER

takes the same options as TF_LOG

Choose where you want to log with **TF_LOG_PATH**

Terraform Log example

```
2021-04-02T15:36:08.477-0400 [DEBUG] provider.terraform-provider-aws_v3.34.0_x5: plugin address:
address=/var/folders/rf/mz_yf4bx0hv1r8m7yqw840jc0000gn/T/plugin730416856 network=unix timestamp=2021-04-02T15:36:08.476-0400
2021-04-02T15:36:08.477-0400 [DEBUG] provider: using plugin: version=5
2021-04-02T15:36:08.525-0400 [TRACE] provider.stdio: waiting for stdio data
2021-04-02T15:36:08.525-0400 [TRACE] GRPCProvider: GetProviderSchema
2021-04-02T15:36:08.593-0400 [TRACE] GRPCProvider: Close
2021-04-02T15:36:08.596-0400 [DEBUG] provider.stdio: received EOF, stopping recv loop: err="rpc error: code = Unavailable desc = transport...
2021-04-02T15:36:08.598-0400 [DEBUG] provider: plugin process exited: path=.terraform/providers/registry.terraform.io/hashicorp/aws/3.34...
2021-04-02T15:36:08.598-0400 [DEBUG] provider: plugin exited
2021-04-02T15:36:08.598-0400 [TRACE] terraform.NewContext: complete
2021-04-02T15:36:08.599-0400 [TRACE] backend/local: finished building terraform.Context
2021-04-02T15:36:08.599-0400 [TRACE] backend/local: requesting interactive input, if necessary
2021-04-02T15:36:08.599-0400 [TRACE] Context.Input: Prompting for provider arguments
2021-04-02T15:36:08.599-0400 [TRACE] Context.Input: Provider provider.aws declared at main.tf:15,1-15
2021-04-02T15:36:08.600-0400 [TRACE] Context.Input: Input for provider.aws: map[string]cty.Value{}
2021-04-02T15:36:08.600-0400 [TRACE] backend/local: running validation operation
2021-04-02T15:36:08.600-0400 [INFO]  terraform: building graph: GraphTypeValidate
2021-04-02T15:36:08.600-0400 [TRACE] Executing graph transform *terraform.ConfigTransformer
```

Commands

Core Workflow	
`terraform init`	It initializes a working directory containing Terraform configuration files. وبستعمله عشان انزل ملفات الproviderاو الmodules.
`terraform plan`	Generates an execution plan, outlining actions Terraform will take. `terraform plan -out=mytfplan ----> Saves the generated plan to a file.`

`terraform apply`	Applies the changes described in the Terraform configuration.
	
	Skips interactive approval prompt (useful in CI/CD).
	`terraform apply --auto-approve`
	
	Sets a variable value directly.
	`terraform apply -var="instance_type=t3.micro"`
	
	Specifies a .tfvars file for variables.
	`terraform apply -var-file="prod.tfvars"`
	
	Applies changes only to a specific resource (use sparingly).
	`terraform apply -target=aws_instance.web`
	
	Uses a custom state file (instead of terraform.tfstate).
	`terraform apply -state=prod.tfstate`
	
	Updates state file with real-world infrastructure changes without modifying resources.
	`terraform apply -refresh-only`

`terraform destroy`	Destroys all resources described in the Terraform configuration. Destroys a specific resource `terraform destroy -target=resource`
State Management	

`terraform state`	Lists all resources in the state.
	`terraform state list`
	
	Shows attributes of a resource.
	`terraform state show aws_instance.web`
	
	Removes a resource from state (does not delete the actual resource).
	`terraform state rm aws_instance.web`
	
	Changes the Terraform resource name from my_server to our_server.
	`terraform state mv aws_instance.my_server aws_instance.our_server`
	
	Moves a resource into a module without recreating it.
	`terraform state mv aws_instance.web module.webserver.aws_instance.web`
	
	Transfers a resource from one state file to another (e.g., during project splits).
	`terraform state mv -state=old.tfstate aws_instance.web -state-out=new.tfstate aws_instance.web`
	
	Downloads the current state file from a remote backend (e.g., S3, Terraform Cloud) to stdout.
	`terraform state pull > terraform.tfstate`
	
	Uploads a local state file to a remote backend.
	`terraform state push terraform.tfstate`

`terraform refresh`	Updates the state file against real resources in the provider. - لو انا بتعامل مع Outputs وكان عندى واحد قديم وبعدين جيت غيرت اسمه مثلا ، الكوماند refresh مش

	هيغيره في الoutputs هو هيضيف اسمه الجديد ويسيب القديم موجود عشان كدة الاحسن انى اعمل plan و apply.

`terraform import`	Imports existing infrastructure into Terraform state.
	`terraform import <resource_address> <resource_id>`
	`terraform import aws_s3_bucket.my_bucket videos_s3_bucket_123`
	`terraform import aws_instance.web i-1234567890abcdef0`

Workspace & Environment	

`terraform workspace`	Creates a new workspace.
	`terraform workspace new dev`
	
	Switches to a workspace.
	`terraform workspace select prod`
	
	Lists all workspaces.
	`terraform workspace list`
	
	Deletes a workspace.
	`terraform workspace delete dev`

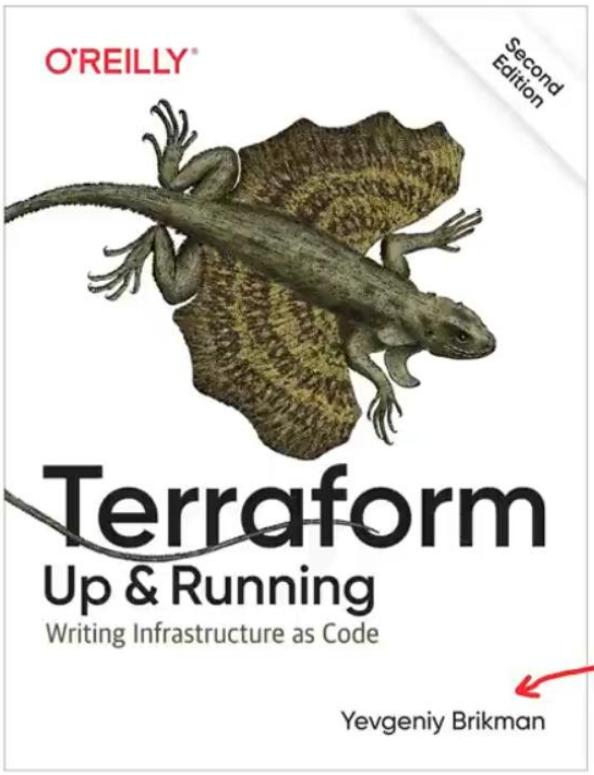
Debugging & Logging	
`terraform validate`	Checks the syntax and validity of Terraform configuration files. بس الكوماند دة مش بيتأكد من ان الvalues بتاعة الattributes صح ولا لا ، بمعنى انى مثلا لو بعمل EC2 الكوماند هيتأكد ان ami attribute موجود والقيمة اللى ادامه string وخلاص ، لكن مش هيتأكد اذا كانت الami صح وموجودة فعلا ولا لا.
`terraform fmt`	Rewrites Terraform configuration files to a canonical format.
`terraform console`	Opens an interactive shell to evaluate expressions.
Advanced & Utility	

`terraform output`	Displays the output values from the Terraform state.
	`terraform output -json ---->Get the output as a json data.`
	`terraform output -raw ---->Get the output as raw data.`
	`terraform output <specific-attribute>`

`terraform providers`	Prints a tree of the providers used in the configuration.
`terraform graph`	Generates a visual representation of the Terraform dependency graph.
`terraform show`	Provides human-readable output from a state or plan file.
`terraform force-unlock`	Manually unlock the state for the defined configuration. `terraform force-unlock LOCK_ID`
`terraform login`	Saves credentials for Terraform Cloud
`terraform logout`	Removes credentials for Terraform Cloud

Additional Resources

Terraform up and running



Terraform Up & Running takes a deep dive into the internal workings of Terraform.

If you want to go beyond this course for things like:

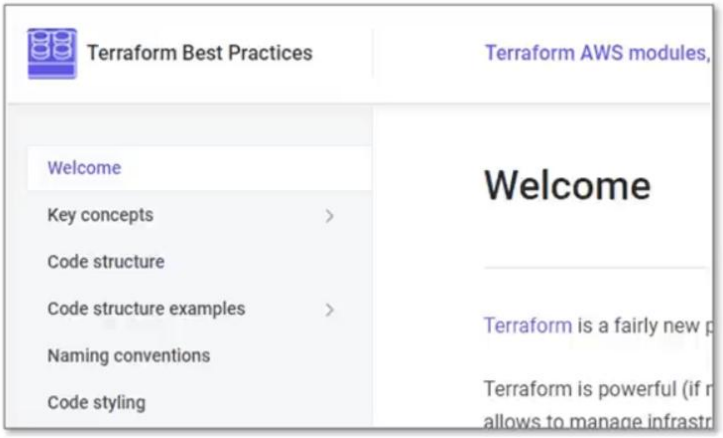
- Testing you Terraform Code
- Zero-Downtime Deployment
- Common Terraform Gotchas
- Composition of Production Grade Terraform Code

Yevgeniy ("Jim") Brikman is the co-founder of **Gruntwork**

Terraform best practices

Terraform is an **online open-book** about **Terraform Best Practices**

www.terraform-best-practices.com



About the Author

Connect with me on Linkedin: mohaned-ahmad

Explore more at: GitHub Repository