**Overview of Terraform**

Terraform is an Infrastructure as Code (IaC) tool that allows you to define and manage cloud infrastructure using a declarative configuration language.

**Example:**

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

**2. Terraform Variables**

Variables in Terraform help make configurations dynamic and reusable.

**Example:**

variable "instance\_type" {

description = "EC2 instance type"

type = string

default = "t2.micro"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = var.instance\_type

}

**3. Terraform Outputs**

Outputs provide useful information about your infrastructure after deployment.

**Example:**

output "instance\_ip" {

value = aws\_instance.example.public\_ip

}

**4. Conditional Expressions**

Allows logic-based conditions within Terraform configurations.

**Example:**

variable "env" {

type = string

default = "dev"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = var.env == "prod" ? "t2.large" : "t2.micro"

}

**5. Terraform Modules**

Modules help break down Terraform configurations into reusable components.

**Example:**

*Module File (main.tf)*

module "network" {

source = "./modules/network"

}

*Module Directory (modules/network/main.tf)*

resource "aws\_vpc" "main" {

cidr\_block = "10.0.0.0/16"

}

**6. Terraform State Management**

State files track the state of infrastructure and should be stored securely.

**Example:**

tfstate show

terraform state list

terraform state rm aws\_instance.example

**7. Terraform Debugging & Troubleshooting**

Use logging and debugging to diagnose issues in Terraform configurations.

**Example:**

export TF\_LOG=DEBUG

terraform apply

**8. Resource Dependencies**

Terraform automatically determines dependencies, but explicit dependencies can be specified using depends\_on.

**Example:**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

depends\_on = [aws\_vpc.main]

}

**9. Terraform Best Practices**

* Use remote state storage (e.g., S3 + DynamoDB for AWS).
* Always use modules for reusable configurations.
* Implement version control for Terraform scripts.
* Use terraform plan before applying changes.

**10. Terraform Data Sources**

Data sources allow Terraform to fetch information from existing resources in your cloud provider.

**Example:**

data "aws\_ami" "latest" {

most\_recent = true

owners = ["amazon"]

}

resource "aws\_instance" "example" {

ami = data.aws\_ami.latest.id

instance\_type = "t2.micro"

}

**Step 1: Define a Data Source**

data "aws\_ami" "latest" {

most\_recent = true

owners = ["amazon"]

}

**Explanation:**

* data "aws\_ami" "latest" → Declares a **data source** to fetch information about an AWS AMI.
* most\_recent = true → Ensures that Terraform selects the latest AMI available.
* owners = ["amazon"] → Filters AMIs to only include those owned by **Amazon** (official AMIs).

👉 **Terraform does not create resources using a data block. Instead, it retrieves existing information from AWS.**

**Step 2: Create an EC2 Instance Using Retrieved AMI**

resource "aws\_instance" "example" {

ami = data.aws\_ami.latest.id

instance\_type = "t2.micro"

}

**Explanation:**

* resource "aws\_instance" "example" → Declares an AWS EC2 instance.
* ami = data.aws\_ami.latest.id → Assigns the **retrieved AMI ID** from the data source.
* instance\_type = "t2.micro" → Specifies a small, cost-effective EC2 instance type.

**Step 3: Apply Terraform Configuration**

Run the following Terraform commands:

**1. Initialize Terraform**

sh

terraform init

**Purpose:** Downloads necessary provider plugins.

**2. Plan the Infrastructure**

sh

terraform plan

**Purpose:** Shows what Terraform will create.

**3. Apply the Configuration**

sh

terraform apply -auto-approve

**Purpose:** Provisions the EC2 instance using the latest AMI.

**Step 4: Verify the Deployed Instance**

Once the instance is created, verify it:

1. **Check in AWS Console:**
   * Navigate to **EC2 Dashboard** → **Instances**.
   * Look for an instance with the **latest AMI**.
2. **Use Terraform Output (Optional)**

output "instance\_public\_ip" {

value = aws\_instance.example.public\_ip

}

sh

terraform apply

This will display the instance's public IP.

**Step 5: Destroy the Resources (If Needed)**

sh

terraform destroy -auto-approve

**Purpose:** Deletes the EC2 instance and releases resources.

**Summary**

✅ **Data Source (data "aws\_ami")** → Fetches the latest AMI  
✅ **EC2 Instance (aws\_instance)** → Uses the latest AMI to create an instance  
✅ **Terraform Commands (init, plan, apply, destroy)** → Deploy & manage infrastructure

**11. Terraform Dynamic Blocks**

Dynamic blocks allow looping inside Terraform resources.

**Example:**

resource "aws\_security\_group" "example" {

dynamic "ingress" {

for\_each = [22, 80, 443]

content {

from\_port = ingress.value

to\_port = ingress.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

This Terraform configuration dynamically creates multiple ingress rules in an **AWS Security Group** using a dynamic block. Below is a step-by-step breakdown:

**Step 1: Define an AWS Security Group**

h

resource "aws\_security\_group" "example" {

* This block defines an **AWS Security Group** named "example".
* A security group controls inbound and outbound traffic to AWS resources (e.g., EC2 instances).

**Step 2: Use a Dynamic Block to Define Multiple Ingress Rules**

dynamic "ingress" { # ✅ 'ingress' is a valid block inside aws\_security\_group

for\_each = [22, 80, 443]

content {

from\_port = ingress.value

to\_port = ingress.value

protocol = "tcp"

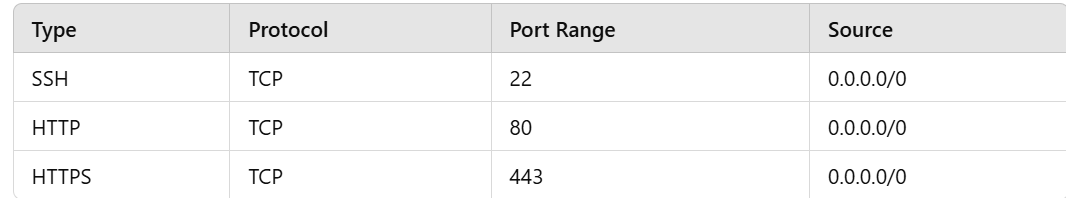
cidr\_blocks = ["0.0.0.0/0"]

}

}

**Explanation:**

1. **dynamic "ingress"** → Creates a **dynamic block** to define multiple **ingress rules**.
2. **for\_each = [22, 80, 443]** → Iterates over the list [22, 80, 443], meaning:
   * Creates a rule for **port 22** (SSH).
   * Creates a rule for **port 80** (HTTP).
   * Creates a rule for **port 443** (HTTPS).
3. **content block** → Defines the actual ingress rule:
   * from\_port = ingress.value → Sets the **starting port** from the list (22, 80, or 443).
   * to\_port = ingress.value → Sets the **ending port** (same as from\_port).
   * protocol = "tcp" → Uses **TCP protocol** for communication.
   * cidr\_blocks = ["0.0.0.0/0"] → Allows traffic from **any IP** (public access).



**Terraform dynamic Blocks - Syntax & Real-Life AWS Examples**

**📌 What is a dynamic Block in Terraform?**

A dynamic block in Terraform allows you to **generate nested configurations dynamically** using loops (for\_each). This is useful when you need to create multiple resources or configurations **without repeating code**.

**📜 General Syntax of dynamic Block**

resource "aws\_resource" "example" {

dynamic "block\_name" {

for\_each = list\_or\_map

content {

# Nested configuration using `block\_name.value`

}

}

}

**🛠 Explanation:**

* dynamic "block\_name" → Declares a dynamic block.
* for\_each = list\_or\_map → Loops through a list or map.
* content { ... } → Defines the actual configuration.

In Terraform, the **block\_name in dynamic "block\_name" must match the expected nested block name of the resource it belongs to**. You **cannot use arbitrary names**.

**💡 Real-Life AWS Examples**

Let's explore practical use cases for Terraform dynamic blocks in AWS.

**Example 1️⃣: Dynamic Security Group Rules**

📌 **Use Case:** Create multiple **ingress rules** in an AWS **Security Group** dynamically.

resource "aws\_security\_group" "web\_sg" {

name = "web-security-group"

description = "Allow inbound traffic for web and SSH"

dynamic "ingress" {

for\_each = [22, 80, 443]

content {

from\_port = ingress.value

to\_port = ingress.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

}

**✅ What Happens?**

* Creates **3 ingress rules** for **SSH (22), HTTP (80), and HTTPS (443)**.
* No need to manually write separate ingress blocks.

**Example 2️⃣: Dynamic EC2 Tags**

📌 **Use Case:** Assign multiple **tags** to an EC2 instance dynamically.

h

resource "aws\_instance" "web\_server" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

dynamic "tag" {

for\_each = {

Name = "WebServer"

Environment = "Production"

Owner = "DevOps Team"

}

content {

key = tag.key

value = tag.value

}

}

}

**✅ What Happens?**

* Adds **3 tags** (Name, Environment, Owner) to the EC2 instance **dynamically**.

**Example 3️⃣: Dynamic IAM Policy Statement**

📌 **Use Case:** Create multiple **IAM policy statements** dynamically.

resource "aws\_iam\_policy" "s3\_access" {

name = "s3-access-policy"

description = "Dynamically generated IAM policy"

policy = jsonencode({

Version = "2012-10-17"

Statement = [

for action in ["s3:GetObject", "s3:PutObject", "s3:DeleteObject"] : {

Effect = "Allow"

Action = action

Resource = "arn:aws:s3:::my-bucket/\*"

}

]

})

}

**✅ What Happens?**

* **Dynamically generates** IAM policy statements for **multiple S3 actions** (GetObject, PutObject, DeleteObject).

**Example 4️⃣: Dynamic EBS Volumes for EC2**

📌 **Use Case:** Attach multiple **EBS volumes** to an EC2 instance dynamically.

resource "aws\_instance" "web\_server" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

dynamic "ebs\_block\_device" {

for\_each = [

{ device\_name = "/dev/sdb", volume\_size = 10 },

{ device\_name = "/dev/sdc", volume\_size = 20 }

]

content {

device\_name = ebs\_block\_device.value.device\_name

volume\_size = ebs\_block\_device.value.volume\_size

}

}

}

**✅ What Happens?**

* Dynamically **attaches 2 EBS volumes** (/dev/sdb - 10GB, /dev/sdc - 20GB) to an EC2 instance.

**Example 5️⃣: Dynamic ALB Listener Rules**

📌 **Use Case:** Create multiple **AWS ALB Listener Rules** dynamically.

resource "aws\_lb\_listener\_rule" "redirect\_rules" {

listener\_arn = aws\_lb\_listener.http.arn

dynamic "condition" {

for\_each = ["example.com", "test.com"]

content {

host\_header {

values = [condition.value]

}

}

}

action {

type = "redirect"

redirect {

port = "443"

protocol = "HTTPS"

status\_code = "HTTP\_301"

}

}

}

**✅ What Happens?**

* Creates **ALB listener rules dynamically** for multiple domains (example.com, test.com).

**🎯 Key Benefits of dynamic Blocks**

1. **Eliminates Code Duplication** → Avoids writing repetitive blocks.
2. **Scalability** → Easily add more items by modifying a list or map.
3. **Flexibility** → Works with various AWS resources like **Security Groups, IAM Policies, EC2, ALB Rules, EBS, and more**.

**🚀 Summary**

| **Use Case** | **AWS Service** | **Dynamic Block Used** |
| --- | --- | --- |
| Security Group Rules | aws\_security\_group | ingress |
| EC2 Tags | aws\_instance | tag |
| IAM Policy Statements | aws\_iam\_policy | Statement |
| EC2 EBS Volumes | aws\_instance | ebs\_block\_device |
| ALB Listener Rules | aws\_lb\_listener\_rule | condition |

**🔹 Now you have a deep understanding of Terraform dynamic blocks and real-world AWS use cases!**

**12. Terraform Provisioners**

Provisioners allow executing scripts on resources during creation or destruction.

**Example:**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "remote-exec" {

inline = [

"sudo apt update",

"sudo apt install -y nginx"

]

}

}

**🚀 Terraform Provisioners - Complete Guide with AWS Examples & Real-Life Projects**

**📌 What are Terraform Provisioners?**

Terraform **Provisioners** allow you to execute scripts or commands on a resource **after it has been created or destroyed**. They help with **configuration management**, **bootstrap scripts**, and **custom deployments**.

**📜 Provisioner Types in Terraform**

Terraform supports **two types** of provisioners:

| **Provisioner Type** | **Description** |
| --- | --- |
| local-exec | Runs commands **on the local machine** where Terraform is executed. |
| remote-exec | Runs commands **on a remote machine (e.g., an EC2 instance).** |

⚠️ **Note:** Provisioners should be used as a last resort. For configuration management, prefer **Ansible, Packer, or cloud-init**.

**📝 Syntax of Provisioners**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo Instance Created!"

}

provisioner "remote-exec" {

inline = [

"sudo apt update -y",

"sudo apt install nginx -y"

]

}

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("~/.ssh/id\_rsa")

host = self.public\_ip

}

}

**🛠 1️⃣ local-exec Provisioner (Runs Locally)**

📌 **Use Case:** Executes a local script or command **on the machine running Terraform**.

**✅ Example: Saving EC2 Public IP to a File**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo ${self.public\_ip} > ec2\_public\_ip.txt"

}

}

**💡 Key Points:**

✔ Runs on **local Terraform host** (not inside the EC2 instance).  
✔ Uses **${self.public\_ip}** to fetch instance details.  
✔ Writes the **public IP** to a local file (ec2\_public\_ip.txt).

**🛠 2️⃣ remote-exec Provisioner (Runs on EC2)**

📌 **Use Case:** Executes remote shell commands **inside the EC2 instance**.

**✅ Example: Install Nginx on EC2**

h

resource "aws\_instance" "web" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "remote-exec" {

inline = [

"sudo apt update -y",

"sudo apt install nginx -y",

"sudo systemctl start nginx"

]

}

connection {

type = "ssh"

user = "ubuntu"

private\_key = file("~/.ssh/id\_rsa")

host = self.public\_ip

}

}

**💡 Key Points:**

✔ Runs commands **inside the EC2 instance**.  
✔ Requires **SSH connection** (user, private\_key, host).  
✔ The inline block runs **multiple commands sequentially**.

**🛠 3️⃣ File Provisioner (Upload Files to EC2)**

📌 **Use Case:** Uploads files or scripts from **local machine to remote EC2 instance**.

**✅ Example: Upload and Execute a Shell Script**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "file" {

source = "setup.sh" # Local file

destination = "/home/ubuntu/setup.sh" # Remote EC2 location

}

provisioner "remote-exec" {

inline = [

"chmod +x /home/ubuntu/setup.sh",

"sudo /home/ubuntu/setup.sh"

]

}

connection {

type = "ssh"

user = "ubuntu"

private\_key = file("~/.ssh/id\_rsa")

host = self.public\_ip

}

}

**💡 Key Points:**

✔ Uploads setup.sh from the **local system** to **EC2**.  
✔ Runs the script **remotely** using remote-exec.

**🛠 4️⃣ Using when = destroy for Cleanup**

📌 **Use Case:** Run commands **before destroying the resource**.

**✅ Example: Deleting a File Before EC2 Destruction**

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "rm -f ec2\_public\_ip.txt"

when = destroy

}

}

**💡 Key Points:**

✔ **Runs only on destroy (when = destroy)**.  
✔ Removes **EC2-related files** before Terraform deletes the resource.

**📌 Points to Remember**

✅ Use **local-exec** when running scripts **on the Terraform host**.  
✅ Use **remote-exec** to execute commands **inside the EC2 instance**.  
✅ **Provisioners are a last resort** → Prefer cloud-init, Ansible, or Packer.  
✅ Ensure **SSH access** is set up before using remote-exec or file.  
✅ Use **when = destroy** for cleanup tasks.

**🌍 Real-Life AWS Projects Using Provisioners**

**🚀 Project 1: Automated Web Server Deployment**

📌 **Goal:** Deploy an EC2 instance with **Nginx and a custom homepage**.  
📌 **Provisioners Used:** remote-exec, file

✔️ **Steps:**  
1️⃣ Create an EC2 instance.  
2️⃣ Upload an **HTML file** to the instance (file provisioner).  
3️⃣ Install and configure **Nginx** (remote-exec).

**Terraform Code:**

resource "aws\_instance" "web" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "file" {

source = "index.html"

destination = "/var/www/html/index.html"

}

provisioner "remote-exec" {

inline = [

"sudo apt update -y",

"sudo apt install nginx -y",

"sudo systemctl restart nginx"

]

}

connection {

type = "ssh"

user = "ubuntu"

private\_key = file("~/.ssh/id\_rsa")

host = self.public\_ip

}

}

✅ **Result:**

* EC2 instance with **Nginx installed**.
* Custom **index.html** uploaded automatically.

**🚀 Project 2: Multi-Server Deployment with Database Setup**

📌 **Goal:** Deploy **two EC2 instances**, one for the **web app** and another for **MySQL**.  
📌 **Provisioners Used:** remote-exec, file

✔️ **Steps:**  
1️⃣ Create a **Web Server (EC2 + Nginx)**.  
2️⃣ Create a **Database Server (EC2 + MySQL)**.  
3️⃣ Upload and execute a **database setup script**.

**13. Terraform Workspaces**

Workspaces allow managing multiple environments (e.g., dev, prod) within the same Terraform configuration.

**Example:**

terraform workspace new dev

terraform workspace select dev

**14. Terraform Import**

Used to import existing infrastructure into Terraform state.

**Example:**

terraform import aws\_instance.example i-1234567890abcdef0

**15. Terraform Destroy**

Removes all resources managed by Terraform.

**Example:**

terraform destroy -auto-approve

**16. Terraform Challenges & Case Studies**

* Handling complex state conflicts.
* Managing large infrastructures with Terraform modules.
* Implementing Terraform in CI/CD pipelines.
* Migrating manual infrastructure to Terraform-managed state.