

Basics

Lecture: Setting up Ubuntu VM on Windows using WSL (Windows Subsystem for Linux)

Purpose of This Lecture

To help **Windows users** quickly set up a **Linux environment (Ubuntu)** using WSL so they can follow the course in a UNIX-like setup, as expected in DevOps, Docker, CI/CD, and containerization environments.

Part 1: Setting up Ubuntu via Microsoft Store

◆ Step-by-step Instructions:

1. Open Microsoft Store

Go to the Microsoft Store on your Windows machine.

Search for “Ubuntu”

In the search bar, type:

nginx
CopyEdit
ubuntu

2.

3. Choose the Distribution

You will see multiple Ubuntu versions:

- Ubuntu (generic, latest LTS)
- Ubuntu 22.04
- Ubuntu 20.04
- Ubuntu 24.04

4. Click on the generic “Ubuntu” option (no version specified) — this usually points to the latest LTS version.

5. Click on "Get" to install

Press the "Get" button to start downloading and installing Ubuntu.

◆ What Happens Next?

- Once installed, click “Open”.

You'll see:

```
css
CopyEdit
This may take a few minutes...
```

- Ubuntu is initializing for first-time use.
-

Username & Password Setup

- You'll be prompted to set:
 - A **username** (e.g., `lauramueller`)
 - A **password** (any password you'll remember)

 **Note:** This password will be needed for commands that require `sudo` (administrator privileges).

Part 2: Verifying the Linux Installation

Once Ubuntu is ready, you can verify your system using these commands:

Command 1: Check Kernel Info

```
bash
CopyEdit
uname -a
```

Purpose: Displays the system information.

 Confirms you're in a Linux kernel environment inside WSL.

 Example output:

```
nginx
CopyEdit
Linux machinename 5.10.102.1-microsoft-standard-WSL2 ...
```

Command 2: Check OS Info

```
bash
CopyEdit
cat /etc/os-release
```

Purpose: Gives details about the installed Linux distribution.

 Output includes:

```
makefile
CopyEdit
NAME="Ubuntu"
VERSION="22.04.4 LTS (Jammy Jellyfish)"
...
```

Part 3: GUI vs CLI Options

If you don't like the default black Ubuntu terminal window from the Store:

Alternate Launch Option: Use PowerShell

Open Windows Terminal or PowerShell

Press **Win + S**, search for:

```
nginx
CopyEdit
terminal
```

1.

Run Ubuntu via WSL

```
bash
CopyEdit
wsl
```

-
2.  This directly opens your Ubuntu virtual environment from within PowerShell.

Understanding Paths and Filesystem Mapping

When running `wsl`, your **Windows and Linux filesystems are connected**.

◆ Path Mapping Example

From within WSL, you might be inside:

```
swift
CopyEdit
/mnt/c/Users/YourName/
```

•

You can switch to your Linux home directory using:

```
bash
CopyEdit
cd ~
```

•

Accessing Ubuntu Files from Windows

To open your Ubuntu files in a graphical explorer window from inside WSL:

```
bash
CopyEdit
explorer.exe .
```

 What it does:

- `explorer.exe`: Calls the Windows Explorer app
- `.:` Represents the current directory

 This command opens your **current Ubuntu Linux folder in Windows Explorer**.



Key Interoperability Highlights

- You can access Linux files from Windows
 - You can launch Windows apps (like Explorer) from Linux WSL
 - You can mount Windows directories inside WSL via `/mnt/c/...`
-



Final Remarks (End of Lecture)

- Ubuntu is now successfully set up on your Windows machine.
 - You're ready to run Linux commands and practice as if you were on a native Ubuntu machine.
 - The system is lightweight, fast, and doesn't require full virtual machines (thanks to WSL).
-



Summary: Tools/Concepts Used

Tool / Concept	Description
Microsoft Store	Platform used to install Ubuntu
Ubuntu	Linux distribution installed
WSL (Windows Subsystem for Linux)	Lightweight Linux environment for Windows
<code>uname -a</code>	Shows kernel/system info
<code>cat /etc/os-release</code>	Displays OS version and metadata
<code>wsl</code>	Command to launch Ubuntu from PowerShell
<code>cd ~</code>	Go to Linux home directory
<code>explorer.exe .</code>	Open current folder in Windows Explorer

Lecture 2: Deep Integration Between Ubuntu (WSL) and Windows

Objective:

To **leverage deeper integration** between your Ubuntu virtual machine (via WSL) and your Windows system:

- Access and manipulate shared files
 - Use Visual Studio Code with WSL
 - Seamless Linux-Windows development experience
 - Setup scripts and manage files from either system
-

Part 1: File Access Between Ubuntu (WSL) and Windows

♦ Starting Ubuntu via Terminal

- Open **any terminal**: PowerShell, Windows Terminal, CMD

Run:

```
bash  
CopyEdit  
wsl
```

•

 This launches your Ubuntu environment **inside the terminal**, with the current working directory starting in the **Windows user path**, e.g.,:

```
swift  
CopyEdit  
/mnt/c/Users/<YourUsername>
```

◆ Listing Files in Windows from WSL

bash

CopyEdit

`ls`

Expected output:

- Desktop
- Documents
- Downloads
- Any file that exists in your Windows user directory

 WSL can **directly list and interact** with your Windows files.

📝 Test: Creating a File in WSL and Accessing It in Windows

Command:

bash

CopyEdit

`echo "hello, world" > hello.txt`

- This creates a file called `hello.txt` in your current WSL directory (which maps to your Windows directory).

Now go to:

makefile

CopyEdit

`C:\Users\<YourUsername>\`

- You'll see `hello.txt` there.

Open it in **Notepad** — you'll see the contents:

CopyEdit

`hello, world`

-

 This proves **file creation in WSL** reflects on the **Windows file system** instantly.

Reverse Test: Deleting File in Windows, Checking in WSL

1. Open `hello.txt` in Windows Explorer
2. Right-click → Delete

Return to WSL and run:

```
bash  
CopyEdit  
cat hello.txt  
Output:
```

```
yaml  
CopyEdit  
cat: hello.txt: No such file or directory
```

- 3.

 File changes/deletions in **Windows** reflect back in **WSL** in real-time.

Visual Linux Filesystem Access via Windows Explorer

Steps:

1. Open **File Explorer**
2. Click "**Linux**" in the sidebar
3. Choose `Ubuntu`

Navigate into:

```
arduino  
CopyEdit  
/home/<your-linux-username>/
```

- 4.

You'll see all your **Linux-side files**, including any files you created inside Ubuntu.

Part 2: Integrating Visual Studio Code with Ubuntu (WSL)

Why do this?

To edit Linux-side files using VS Code on Windows **without breaking file permissions or paths**, and to run Linux shell commands inside VS Code terminal.

Setup Steps:

1. Open VS Code → Click on Extensions panel (left sidebar)

2. Search:

nginx

CopyEdit

wsl

3. Install:

 **Windows Subsystem for Linux (WSL)** extension by Microsoft

Remote Explorer Integration

Once WSL extension is installed:

1. Open the **Remote Explorer** tab (left sidebar)
2. Click on **Ubuntu**

Choose:

sql

CopyEdit

Connect in Current Window

- 3.

 This opens a **VS Code session inside WSL**, connected directly to your Ubuntu file system.

Test: Launching a Linux Terminal in VS Code

1. Open terminal (**Ctrl + backtick**)

You'll see:

```
ruby
CopyEdit
user@ubuntu:~$
```

- 2.

 You're **running a Linux terminal inside VS Code**, operating within WSL Ubuntu.

Part 3: Creating & Executing a Shell Script Inside WSL + VS Code

Step-by-Step in WSL

Navigate to your home directory (just in case):

```
bash
CopyEdit
cd ~
```

- 1.

Create a directory:

```
bash
CopyEdit
mkdir dummy
cd dummy
```

- 2.

Create a script file:

```
bash
CopyEdit
echo "echo hello world" > say-hi.sh
```

3.

Make it executable:

```
bash
CopyEdit
chmod +x say-hi.sh
```

4.

Run the script:

```
bash
CopyEdit
./say-hi.sh
```

5.

Expected output:

```
nginx
CopyEdit
hello world
```

Open Dummy Folder in VS Code

Inside dummy directory:

```
bash
CopyEdit
code .
```

This opens the **current folder** (dummy) in VS Code. Inside:

- You'll see your script (`say-hi.sh`)
- You can edit the shell script graphically

 **Note:** You're still connected to your **Ubuntu VM**, so all terminals and files stay within WSL.

Key Features & Benefits Recap

Feature	Benefit
wsl command	Instantly boot Linux VM in terminal
File reflection	Changes in Linux ↔ Windows sync in real-time
VS Code WSL Extension	GUI development with terminal and shell integration
Run Windows apps from Linux	e.g., <code>explorer.exe .</code>
Run Linux commands from Windows	Seamless development flow

Commands Used in Lecture

```
bash
CopyEdit
# Open WSL (Ubuntu)
wsl

# List files
ls

# Create file with echo
echo "hello, world" > hello.txt

# Delete file in Windows → test from WSL
cat hello.txt

# Navigate to home
cd ~

# Create directory
mkdir dummy && cd dummy

# Create shell script
echo "echo hello world" > say-hi.sh

# Make script executable
chmod +x say-hi.sh
```

```
# Run script  
./say-hi.sh  
  
# Open current folder in VS Code  
code .
```



Final Notes from Instructor

- **Visual Studio Code + WSL** is **highly recommended** for best experience
 - Instructor is **actively revising** lectures to ensure full compatibility with this setup
 - If any problems arise, students are encouraged to **ask in the Q&A section**
 - All tools, packages, and commands moving forward will assume you're working inside the **Ubuntu WSL VM**
 - When installing anything, always follow **Linux instructions**, not Windows ones
-



You Now Have:

- A Linux VM installed and integrated inside Windows
- Cross-platform file access
- GUI + shell toolset (VS Code + Bash)
- Environment ready for DevOps, Docker, Terraform, CI/CD training

Lecture 3: AWS CLI Authentication via Environment Variables (Local Setup)

Objective:

To **authenticate your local development environment** (Ubuntu WSL) with AWS using an **IAM user's access key and secret**. This enables you to run AWS CLI commands (and tools like Terraform) that interact with AWS services like S3, EC2, etc.

Step-by-Step Breakdown

Step 1: Confirm You Are Not Yet Authenticated

Command:

```
bash  
CopyEdit  
aws s3 ls
```

 What it does:

- Tries to **list all S3 buckets** for the currently authenticated AWS identity (if any).

 Expected output (error):

```
vbnnet  
CopyEdit  
Unable to locate credentials. You can configure credentials by  
running "aws configure".
```

 Confirms you haven't configured AWS credentials yet.

Step 2: Create a New IAM Access Key in AWS Console

 Navigation Path in AWS Console:

1. Open AWS Console
 2. Go to **IAM (Identity and Access Management)**
 3. Click on “**Users**”
 4. Select your user (e.g., **UdemyAdmin**)
 5. Click on “**Security Credentials**” tab
 6. Scroll to the “**Access Keys**” section
-

Create a New Access Key:

- Click: **Create access key**
 - Choose:
 - Use case** → *Command Line Interface (CLI) or Local Code*
 - It doesn't really matter which you pick — purely informational)
 - Proceed:
 - Click **Next**
 - Acknowledge: "**I understand the recommendations**"
 - Give a **description** (e.g., **Terraform Course**)
 - Click **Create Access Key**
-

You'll now see:

- **Access Key ID**
- **Secret Access Key**

 Copy both **immediately** — the secret is **only shown once**.



Step 3: Save Keys as Environment Variables Locally

◆ **AWS Documentation Reference:**

Environment variables for AWS CLI authentication:

```
bash
CopyEdit
AWS_ACCESS_KEY_ID=your_key_id
AWS_SECRET_ACCESS_KEY=your_secret
AWS_DEFAULT_REGION=your_region (optional)
```

 **Step 4: Store These in a .env File (Best Practice)**

 **In Your Project Folder:**

Create a new file called:

```
bash
CopyEdit
.env
```

Paste the following into it:

```
env
CopyEdit
AWS_ACCESS_KEY_ID=your_actual_access_key_id
AWS_SECRET_ACCESS_KEY=your_actual_secret_key
# AWS_DEFAULT_REGION=us-east-1 # Optional
```

 **Important Cleanup:**

- **Remove** any \$ signs from variable values if present (from docs).
-

 **Step 5: Create a .gitignore to Avoid Credential Leaks**

Create:

```
CopyEdit
.gitignore
```

Add:

```
bash  
CopyEdit  
.env
```

-  This ensures you **don't accidentally push your secret credentials to GitHub or any remote repo.**
-

Step 6: Load Your Environment Variables into Current Terminal

Run:

```
bash  
CopyEdit  
source .env
```

This will export your AWS credentials into the **current shell session**.

 **Note:** This works **only for the current terminal**.

- If you open a new terminal, you'll need to run `source .env` again.
-

Step 7: Re-Test the AWS Command

Run again:

```
bash  
CopyEdit  
aws s3 ls
```

-  If successful, you'll now see a list of your S3 buckets (or none, if your account is new).

-  Green arrow icon = successful command
 -  Red arrow icon = failure (in the instructor's IDE, but behavior may vary)
-

Important Concepts & Best Practices

Concept

Description

<code>aws s3 ls</code>	Lists S3 buckets for the authenticated AWS identity
<code>.env</code> file	Stores your AWS credentials temporarily and securely
<code>source .env</code>	Loads environment variables into current shell session
<code>.gitignore</code>	Prevents sensitive files (like <code>.env</code>) from being pushed to git
Temporary Scope	Credentials from <code>.env</code> last only for that session
<code>AWS_ACCESS_KEY_ID,</code> <code>AWS_SECRET_ACCESS_KEY</code>	Core credentials needed for CLI access

⚠️ Security Reminder from Instructor

- Keys shown in the video are for **demo purposes only**
- Instructor deletes them immediately after
- 🔒 **Never expose or share your AWS credentials**

📦 Commands Used in This Lecture

```

bash
CopyEdit
# Check if credentials are set
aws s3 ls

# Exporting credentials manually (alternative)
export AWS_ACCESS_KEY_ID=your_access_key
export AWS_SECRET_ACCESS_KEY=your_secret_key

# Load env variables from .env file
source .env

# Check again
aws s3 ls

```

After This Lecture, You Have:

- An authenticated CLI setup to interact with AWS
 - Secrets **stored securely** via `.env`
 - Git-safe config via `.gitignore`
 - Local dev machine **ready to deploy infrastructure** (e.g., with Terraform, AWS CLI, etc.)
-

Next Steps

Now that you're authenticated, the next lecture will likely begin provisioning AWS infrastructure (probably using Terraform or CLI).

Intro to IAC

Lecture 4 Notes: What is Infrastructure as Code (IaC)? Why Terraform?

What is Infrastructure as Code (IaC)?

- **Definition:**

Infrastructure as Code is a method of **provisioning and managing infrastructure using code**, rather than clicking manually on a cloud provider's console or using ad-hoc CLI commands.

The "Before IaC" Problem

1. Manual Approach

- Using cloud **console** (UI): click-through setup
- Using **CLI/scripts**: command-based setup
- Both require knowing:
 - **Where to click / what values to fill in** (console)
 - **Precise commands and parameters** (CLI)

2. Problems with Manual + CLI Setup

Problem	Description
 Hard to track	No history of who did what
 Breaking changes	Easy for someone to accidentally break infra
 Poor reproducibility	Re-creating large infra (VPC, EC2, DBs, Gateways) is tough
 Imperative logic	You must know the <i>how, what, and in what order</i>
 Error-prone	One wrong step or misconfig = failure in production
 Poor auditing	No versioning, no easy rollback or team collaboration

IaC: The Solution

- **Code defines infrastructure** → You describe the **desired state**, not step-by-step commands.
 - You use **declarative configuration files**, not manual steps.
 - IaC tools (like **Terraform**) interpret your code and:
 - Compare it to the current infra
 - Generate a **plan of changes**
 - Apply those changes **only if you approve**
-

IaC Advantages

Benefit	Description
 Reproducibility	Easily replicate complex infra anywhere
 Version control	Code lives in Git – track, revert, audit
 CI/CD Ready	Automate deployment via CI/CD pipelines
 Validation	Add input validation, prevent misconfigurations
 Declarative > Imperative	No need to know every API detail or command
 Team-friendly	Changes are visible, reviewable, and trackable

Declarative vs Imperative

Declarative	Imperative
<i>What</i> to build	<i>How</i> to build it
Infra described in end-state code	Sequence of manual steps
Easier to maintain and audit	Harder to reason about at scale
Terraform-style	Bash scripts, CLI, manual

What's Coming Next

- Terraform will be used as the **primary IaC tool**.
 - Upcoming lectures will dive deep into:
 - How Terraform tracks resources
 - How it communicates with cloud APIs
 - Full resource lifecycle management
-

Summary

- IaC transforms infra management from manual chaos to **codified clarity**.
- Tools like Terraform make infra:
 - **Repeatable**
 - **Versioned**
 - **Safer to deploy**
 - **Collaborative**

Lecture 2 Notes: Key Benefits of Infrastructure as Code

1. Better Cost Management

- **Easy creation/destruction of infra**
 - Everything is codified → one command to **spin up / tear down** environments
 - Example: Spin up **dev environments** during work hours, destroy at night to save cost
- **Automation saves time**
 - Devs and ops teams can focus on higher-value tasks
- **Tagging strategies**
 - Enforce and automate **resource tagging**
 - Helps with **billing visibility**, categorization, and audits
- **Infra visibility**
 - Clear overview of all resources managed by the project

2. Improved Reliability

- **Consistent deployment behavior**
 - IaC tools interact **consistently** with cloud APIs
 - Reduces variability from manual actions
- **Multiple deployment modes**
 - Local, CI/CD, or API-triggered — same reliable outcome
- **Built-in validation**
 - IaC tools often catch **invalid values/configs** before deployment

3. Improved Consistency & Scalability

- **Reusable code & modules**
 - Common infra (e.g., networking, security) can be modularized
 - Use same templates across environments (dev, staging, prod)
 - **Easy infra duplication**
 - Deploy same setup across regions/projects
 - **Scalable resource counts**
 - Increase/decrease infra size (e.g., number of instances) via config
-

4. Improved Deployment Process

- **Automation reduces friction**
 - Deployments become **repeatable and faster**
 - **Prevents config drift**
 - If someone manually changes infra outside of code:
 - IaC tools will **detect and revert** the unexpected changes
 - **Integrated with CI/CD**
 - Infra changes are just part of the deployment pipeline now
 - No more separate, manual "infra" step
 - **Version controlled**
 - Easier to **revert to known good state** after bad deployments
-

5. Fewer Human Errors

- **Plan stage preview**
 - IaC shows **all planned changes** before applying them
 - **Use of variables and references**
 - Avoids hardcoded values
 - Connect resources more cleanly and reliably
 - **Custom validations**
 - Add checks for valid inputs, proper configs
 - **Protection rules**
 - Prevent accidental deletion of critical resources
-

6. Improved Security Strategies

- **Compliance enforcement**
 - Example: Ensure **EBS volumes are encrypted**
 - Example: Ensure **EC2 uses pre-approved AMIs**
 - **Secure shared modules**
 - Security-focused teams can maintain shared infra modules
 - Ensures best practices are baked in
 - **Manage IAM users, roles, and policies**
 - All done via code → easy to audit and review
 - **Static security scanning**
 - IaC code can be scanned for **vulnerabilities** just like app code
-

7. Self-Documenting Infrastructure

- **Code = Documentation**
 - The infra code *is* the spec of what gets deployed
 - **List & inspect resources**
 - IaC tools track which resources were created and how
 - **CI/CD logs available**
 - Logs from pipeline runs help trace and debug issues
-

Summary

Infrastructure as Code brings:

- **Automation**
- **Version control**
- **Security**
- **Auditing**
- **Error prevention**
- **Faster scaling**
- And much more.

It is now the **industry standard** for managing infrastructure.

Lecture 6: Creating AWS VPC Manually — Step-by-Step Networking Demo

Goal of the Demo

Manually create:

- A **VPC** (`demo VPC`)
 - Two **subnets**:
 - **Public subnet** (internet access)
 - **Private subnet** (no internet access)
-

Step-by-Step: Create a VPC

1. **Go to AWS Console → Set desired region**
[You must choose your preferred AWS region]
2. **Search for “VPC”**
 - VPC = *Virtual Private Cloud*
 - Logical, isolated network for resources
3. **View existing VPCs**
 - Every region has a default VPC and subnets
 - Default VPC = pre-configured by AWS
4. **Create new VPC:**
 - Click: “**Create VPC**”
 - Option: **VPC only**
 - Name: `demo VPC`

- IPv4 CIDR block: `10.0.0.0/16`
(Allows large IP range — up to 65,536 addresses)
- No IPv6
- Tenancy: Default
- Click: **Create VPC**

✓ Result:

- VPC created
 - A **main route table** is automatically created with it
-

🌐 Step 2: Create Subnets

1. Go to “Subnets” → **Create Subnet**

- VPC: `demo VPC`

🟢 Public Subnet

- Name: `public subnet`
- AZ: Any (e.g., `us-east-1a`)
- CIDR block: `10.0.0.0/24`

🔒 Private Subnet

- Click **Add new subnet**
- Name: `private subnet`
- AZ: Any (e.g., `us-east-1b`)
- CIDR block: `10.0.1.0/24`

📌 /24 subnet:

- Leaves first 3 octets fixed (e.g., `10.0.0.X`)
- Final octet ranges for addresses
- Helps prevent CIDR overlap

 Result: Two subnets created under your custom VPC

Step 3: Add Internet Access (Public Subnet)

1. **Go to “Internet Gateways”**
 - Default IGW exists for default VPC
2. **Create new IGW**
 - Name: `demo-igw`
 - Click: **Create**
 - Initial status: `Detached`
3. **Attach IGW to VPC**
 - Click **Actions → Attach to VPC**
 - Choose: `demo VPC` → Attach

Step 4: Route Tables Setup

1. **Go to “Route Tables”**
 - Main route table exists for every VPC
 - It applies **implicitly** to any subnet not explicitly associated
2. **Create a new Route Table**
 - Name: `public route table`

- VPC: `demo` VPC
- Click: **Create Route Table**

3. Add Internet Route

- Click: **Edit Routes**
- Add route:
 - Destination: `0.0.0.0/0` (catch-all)
 - Target: Select `demo-igw`
- Click: **Save changes**

 Result:

- `public route table` now has:
 - Local route
 - Public internet route via IGW

Step 5: Associate Subnets with Route Table

1. Select `public route table` → **Subnet Associations**
2. Click: **Edit subnet associations**
 - Select: `public subnet`
 - Click: **Save associations**

 Outcome:

- **Public subnet** explicitly linked to **internet-connected route table**

Verifying Route Table Associations

Public Subnet:

- Explicitly associated with `public route table`
→ Has internet access

Private Subnet:

- Not explicitly linked
→ Uses **main route table by default**
→ **No IGW → No internet access**
-



Wrap-up: Why Manual Setup Isn't Ideal



Key insight:

- Even for a **basic VPC setup**, many steps + details (CIDRs, AZs, routes, tags, associations)
- Easy to misconfigure something
- Not scalable or reliable for **production use**



Takeaway:

Manual infrastructure = for learning

But for real-world reliability



Motivation to adopt **Infrastructure as Code (IaC)** — where all of this is:

- Automated
- Version-controlled
- Repeatable
- Scalable

Lecture 7: First Look at Terraform – Recreating Our VPC Setup as Code

Goal

Replicate the previous manual VPC setup using **Terraform**, an Infrastructure as Code (IaC) tool:

- One VPC
- Two subnets (public and private)
- Internet Gateway
- Route Table + association

 *This is only a preview. Don't worry if the syntax doesn't make full sense yet — it'll be explained step-by-step later.*

Step 1: Create Project Folder and File

- Folder: `01-benefits`
- File: `vpc.tf`
- All Terraform files use `.tf` extension and **HCL** (HashiCorp Configuration Language)

Step 2: Terraform Boilerplate

h

CopyEdit

```
terraform {  
    required_providers {  
        aws = {
```

```
    source  = "hashicorp/aws"
    version = "~> 5.0"
  }
}

}
```

- Declares required provider (`aws`) from official HashiCorp source
 - Version constraint: accepts `5.x` versions
-

Step 3: Create AWS Resources in Code

1. VPC

hcl

CopyEdit

```
resource "aws_vpc" "demo_vpc" {
  cidr_block = "10.0.0.0/16"
}
```

- Same CIDR block as manual setup
- Name: `demo_vpc`

2. Public Subnet

hcl

CopyEdit

```
resource "aws_subnet" "public_subnet" {
  vpc_id      = aws_vpc.demo_vpc.id
```

```
    cidr_block = "10.0.0.0/24"  
}  
  

```

3. Private Subnet

hcl

CopyEdit

```
resource "aws_subnet" "private_subnet" {  
  
    vpc_id      = aws_vpc.demo_vpc.id  
  
    cidr_block = "10.0.1.0/24"  
  
}
```

 Note: Using `.id` references the `id` of the previously declared resource.

Step 4: Internet Gateway

hcl

CopyEdit

```
resource "aws_internet_gateway" "igw" {  
  
    vpc_id = aws_vpc.demo_vpc.id  
  
}
```

Step 5: Public Route Table

hcl

CopyEdit

```
resource "aws_route_table" "public_rtb" {
```

```
vpc_id = aws_vpc.demo_vpc.id

route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
}
}
```

- Default route for all traffic to use the Internet Gateway
-

Step 6: Associate Subnet to Route Table

hcl

CopyEdit

```
resource "aws_route_table_association" "public_assoc" {
    subnet_id      = aws_subnet.public_subnet.id
    route_table_id = aws_route_table.public_rtb.id
}
```

Step 7: Provider Configuration

hcl

CopyEdit

```
provider "aws" {
    region = "eu-west-1"  # Replace with your AWS region
```

}

Summary: Benefits of Terraform

- Cleaner, declarative format
 - All dependencies (VPC → Subnet → IGW → RTB) resolved **automatically**
 - No clicking around — **repeatable** and **version-controlled**
 - Easy to connect resources by referencing `.id`
-

Want to Try It?

- This is a **fully working Terraform config**
 - You can test it with:
 - `terraform init`
 - `terraform apply`
 - But in this course, we'll **learn each concept before using them**
-

Takeaway

Terraform makes complex infrastructure **clear, connected, and reproducible**.

This preview shows how powerful **Infrastructure as Code** is compared to manual setup — a great reason to dive deeper into it.

Lecture 4 – Why Use Terraform Over Other IaC Tools

Overview

This lecture answers the **frequent question**:

“Why Terraform over other Infrastructure as Code (IaC) tools like AWS CloudFormation or Pulumi?”

Key Advantages of Terraform

① Platform Agnostic (Multi-Cloud & On-Prem Support)

- **Terraform is not tied to a single cloud provider** (unlike CloudFormation which is AWS-only).
- Can manage infrastructure for:
 - AWS, Azure, GCP
 - On-prem providers (e.g., VMware, OpenStack)
 - SaaS services (e.g., GitHub, Datadog, Cloudflare)
- All these **can coexist in a single Terraform configuration/project**.

Use case example:

Spin up a web server on AWS, configure DNS in Cloudflare, and set monitoring in Datadog — **all in one Terraform file**.

② High-Level Abstraction

- Terraform is **not tightly coupled** to a platform — it gives a **cleaner abstraction** layer over multiple providers.
- Great when:
 - Teams work across clouds
 - You need **vendor neutrality**

- You want to **avoid cloud lock-in**
-

③ Modular Architecture (Reusable Modules)

- Terraform supports **modularity natively**:
 - Define a group of related resources as a **module**
 - Reuse the same module with different parameters
 - Combine small modules into **larger architectures**

📦 Example:

hcl

CopyEdit

```
module "vpc" {  
  
  source = "./modules/vpc"  
  
  cidr_block = "10.0.0.0/16"  
  
}
```

➡ Cleaner code, easier team collaboration, scalable infra

There's an entire upcoming section dedicated to modules.

④ Parallel Deployment via Dependency Graph

- Terraform **auto-generates a DAG (Directed Acyclic Graph)** of resources.
- It determines:
 - Which resources depend on others
 - Which resources can be created **in parallel**

- Which must be **created sequentially**

⚡ Result: **Fast deployments & efficient infra changes**

5 Plan Before Apply (Safe Infra Changes)

- Terraform separates **planning and execution** phases:
 - `terraform plan` → **shows what will change**
 - `terraform apply` → **actually makes the change**

🛡 Helps:

- Review changes before they happen
 - Prevent surprises or unintended deletions
-

6 Resource Protection & Validation

- Protect against **accidental deletes/changes** using:
 - `lifecycle { prevent_destroy = true }`
- Add **input/output validations** using:
 - `validation` blocks in variables

✓ Enforces safe & intentional infrastructure changes

7 Terraform State File = Performance Booster

- State file = **Terraform's memory** of what's deployed.
- Binds **local configuration ↔ real-world infra**
- Enables:

- Faster plans & applies
- Easier detection of drift
- Optimized operations

📁 State is stored locally or remotely (e.g., S3 with locking via DynamoDB).

🧠 Full **state management** will be discussed in detail in future lectures.

🏁 Summary – Why Terraform Wins

Feature	Terraform ✓	Others (like CloudFormation, Pulumi) ✗
Multi-Cloud Support	✓	✗ CloudFormation (AWS-only)
Abstraction Layer	✓	✗ Often tightly coupled
Modules & Reusability	✓	Partial
Parallel Execution	✓	✗ CloudFormation = slow
Safe Planning & Applying	✓	Often limited
Resource Protection	✓	Partial
State File Optimization	✓	✗ or hidden behind abstraction

Lecture: Terraform Architecture, Providers & Registry

How Terraform Works – Overview

- Terraform uses a **plugin-based architecture** to manage infrastructure across multiple platforms (AWS, GCP, Azure, etc.).
- At the core of Terraform is the **Terraform Core** binary, which:
 - Reads and interprets `.tf` files.
 - Maintains the state.
 - Performs graph operations and plan/apply logic.

Providers – The Plugin System

- **Providers** are external plugins used by Terraform to interact with cloud platforms or services.
- Each provider is a separate binary (written to follow a specific interface defined by Terraform Core).
- They are responsible for **CRUD** operations (Create, Read, Update, Delete) on infrastructure resources.

 Examples:

- `aws` provider = for AWS APIs.
- `google` = for GCP APIs.
- `azurerm` = for Azure APIs.

 Anyone can write a provider if they follow Terraform's provider plugin **interface contract**.

Why Declare Providers in `.tf`?

- Terraform doesn't include all provider logic by default.

In every project, you **must declare** the providers you're using in the configuration:

```
hcl
CopyEdit
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

- During `terraform init`, these providers are **downloaded from the Terraform Registry**.
-

Terraform Registry – The Provider & Module Hub

📍 URL: <https://registry.terraform.io>

Types of Components:

1. **Providers** – Allow interaction with cloud APIs.
2. **Modules** – Reusable components (we'll explore later).

Provider Tiers:

- **Official**: Maintained by HashiCorp.
- **Partner**: Maintained by 3rd parties, verified by HashiCorp.
- **Community**: Unverified, inspect carefully for security & maintenance status.

Using the AWS Provider:

- Navigate to AWS provider page → Click "**Use Provider**".
 - You'll see the snippet required to declare it in your `.tf` file.
 - Also includes a **link to documentation** for all supported AWS resources.
-



Provider Docs – Resource Reference

- Every resource (e.g., `aws_vpc`, `aws_instance`) has:
 - **Usage examples**.
 - **Argument list** – required/optional attributes.
 - **Exported attributes** – values accessible after creation (e.g., `vpc.id`, `cidr_block`).

Example:

- Navigate to `aws_vpc` in the AWS provider docs.
 - You'll find:
 - `cidr_block`, `tags`, `enable_dns_support`, etc.
 - Real-world HCL examples for VPC creation.
-



Summary

- Terraform is modular and extensible via **providers**.
- These plugins handle communication with external APIs.
- The **Terraform Registry** is your go-to place to:
 - Discover providers.
 - Read docs and examples.

- Browse reusable modules.
 - When stuck, **always check the official documentation** for the resource or provider.
-

 Up Next: We'll start exploring resource blocks and write real Terraform code for provisioning infrastructure using these providers.



Terraform Lifecycle Phases: Plan, Apply, Destroy

Overview:

Terraform manages infrastructure via **three key lifecycle phases**:

1. **Plan** – Preview changes
 2. **Apply** – Make changes
 3. **Destroy** – Tear down infrastructure
-

Inputs to Terraform:

- **Configuration Files** (e.g., `main.tf`, `variables.tf`)
- **Terraform State File** (e.g., `terraform.tfstate`)
 - Keeps track of existing infrastructure resources

Terraform uses these **as input** to understand:

- What should exist (based on configs)
 - What actually exists (based on state and provider APIs)
-

Phase 1: `terraform plan`

Purpose:

- Compares:
 - Desired infrastructure (`.tf` config files)
 - Current state file
 - Real-world infrastructure (via providers)

What Happens:

1. Terraform contacts provider APIs (e.g., AWS, Azure)
2. Fetches **actual state of resources**
3. Compares:
 - Actual state (from API)
 - Stored state file
 - Desired state (from code)
4. Outputs an **execution plan**:
 - What will be **created, modified, or deleted**

Example output:

bash

CopyEdit

```
~ aws_instance.example

    ami: "ami-abc123" => "ami-def456"

+ aws_s3_bucket.mybucket

    name: "my-new-bucket"
```

 This plan is only a dry run – no changes made yet.

Phase 2: **terraform apply**

Purpose:

- Executes the plan and **applies changes** to infrastructure

What Happens:

- Terraform:
 - Re-reads provider API states
 - Executes **create / update / delete** operations as per plan
- New infra state is saved into `terraform.tfstate`

Notes:

- You can skip running `terraform plan` separately.

If you run:

```
bash
CopyEdit
terraform apply
```

- without a plan, Terraform internally runs **plan first**, then prompts for confirmation.
-

💣 Phase 3: `terraform destroy`

Purpose:

- Completely removes all infrastructure tracked by Terraform

When to use:

- For **ephemeral** environments (e.g., dev/test)
- When cleaning up unused projects to **save costs**

What Happens:

- Terraform reads from state file
- Uses providers to **delete every managed resource**

Example:

```
bash
```

CopyEdit

`terraform destroy`

 **Dangerous command** – no partial delete; deletes *everything managed*.

! Difference Between Delete vs Destroy:

Scenario	Action
You delete a resource from <code>.tf</code> file	Terraform deletes just that resource during next <code>apply</code>
You run <code>terraform destroy</code>	Terraform deletes everything in the state

Summary:

Phase	Description
<code>plan</code>	Compares desired config with real world infra; outputs intended changes
<code>apply</code>	Executes plan; creates/updates/deletes resources
<code>destroy</code>	Wipes out all managed infrastructure



Hands-On: Running a Full Terraform Lifecycle

📁 Project Structure

- Reusing a `.tf` file from earlier (VPC creation example).
 - Resources included:
 - `aws_vpc`
 - `aws_internet_gateway`
 - `aws_route_table`
 - `aws_route_table_association`
 - `aws_subnet` (private & public)
-

✓ 1. Terraform Initialization (`terraform init`)

When to run:

- First time in a project
- After adding/changing a **provider** or its **version**

What it does:

- Downloads provider plugins (e.g., AWS)
- Creates a `.terraform/` folder
- Generates a `terraform.lock.hcl` file to **pin provider versions + hashes**

Output includes:

bash

CopyEdit

Terraform has been successfully initialized!



2. Planning Changes (`terraform plan`)

Before running:

- Authenticate with AWS by sourcing the `.env` file:

bash

CopyEdit

```
source .env
```

- Then run:

bash

CopyEdit

```
terraform plan
```

What happens:

- Terraform contacts AWS using provider credentials
- Compares:
 - Real infrastructure (via AWS API)
 - Local state file
 - `.tf` config files

Output (example):

cpp

CopyEdit

Terraform will perform the following actions:

```
+ aws_vpc.main  
+ aws_internet_gateway.main  
+ aws_route_table.main  
+ aws_subnet.public  
+ aws_subnet.private  
+ aws_route_table_association.public
```

Note:

- The list is **not in execution order** – Terraform auto-manages the correct **resource dependency graph** internally.
-



Tagging for Clarity:

In the config, a tag was added:

hcl

CopyEdit

```
tags = {  
  Name = "Terraform VPC"  
}
```

- Helps visually identify the resource in the AWS console.
-



3. Applying Changes (`terraform apply`)

- Run directly:

bash

CopyEdit

```
terraform apply
```

- If no prior plan is given:
 - Terraform internally runs `plan` again.

Then asks for confirmation:

pgsql

CopyEdit

```
Do you want to perform these actions?
```

- Type `yes` to proceed.

Execution:

- Terraform creates resources in dependency order:
 1. VPC
 2. Internet gateway & subnets (in parallel)
 3. Route table
 4. Route table association

Output:

bash

CopyEdit

```
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
```

Verification:

- In AWS Console (switch to correct region):

- Go to **VPC Dashboard**
 - See the new VPC named "**Terraform VPC**"
-



4. Destroying Infrastructure (`terraform destroy`)

- Run:

bash

CopyEdit

`terraform destroy`

What happens:

- Terraform:
 - Refreshes state (checks current live infra)
 - Plans to destroy all 6 managed resources

Prompts for confirmation:

`pgsql`

CopyEdit

`Do you really want to destroy all resources?`

- Type `yes`

Output:

bash

CopyEdit

`Destroy complete! Resources: 6 destroyed.`

Console:

- After refresh → the VPC and all related resources are **gone**
-

⚠ Caution: **destroy** vs **apply**

Action	Use case
<code>terraform destroy</code>	Deletes all resources tracked in state file
<code>terraform apply</code>	Add/update/delete only the necessary changes

Destroy is useful for:

- **Ephemeral environments** (e.g., dev/test)
- Full cleanup before a project reset

But in production, it's safer to use **apply**.

🧠 Summary

Command	Purpose
<code>terraform init</code>	Initialize project, download providers
<code>terraform plan</code>	Preview infrastructure changes

`terraform apply` Execute infrastructure changes

`terraform destroy` Wipe all resources managed by Terraform

HCL Intro

Lecture Notes — HCL Overview (Terraform Language Syntax)

Purpose of This Lecture:

- Understand the **syntax and structure** of HashiCorp Configuration Language (HCL).
 - Explore the **main blocks** used in Terraform.
 - Not about wiring a complete working setup — focus is on **learning syntax**, not execution.
-

Project Setup

Create a new folder:

```
bash
CopyEdit
mkdir 02-hcl
cd 02-hcl
```

•

Create a new file (name is arbitrary):

```
bash
CopyEdit
touch hcl.tf
# or main.tf, config.tf, etc.
```

•

1. **terraform** Block

Used to configure the project's **Terraform settings**:

```
hcl
CopyEdit
terraform {
  required_providers {
```

```
aws = {
  source = "hashicorp/aws"
  version = "5.37.0"
}
}
```

- Defines **provider source** and **version**.
 - Important for pinning versions and ensuring consistent behavior across environments.
-

2. resource Block

Used to **create and manage infrastructure**:

```
hcl
CopyEdit
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-sample-bucket"
}
```

- ◆ Format: **resource "<TYPE>" "<LABEL>" { ... }**
 - **TYPE**: e.g., `aws_s3_bucket`, `aws_vpc`, etc.
 - **LABEL**: a unique name within that type.
 - Block content varies by type — arguments are resource-specific.

Example with duplicate labels (⚠ invalid):

```
hcl
CopyEdit
resource "aws_s3_bucket" "my_bucket" { ... }
resource "aws_s3_bucket" "my_bucket" { ... } # ✗ Not allowed
```

3. data Block

Used to **reference external infrastructure** not managed by Terraform:

```
hcl
CopyEdit
data "aws_s3_bucket" "my_external_bucket" {
  bucket = "not-managed-by-us"
}
```

- ◆ Purpose:
 - Lookup infrastructure (e.g. existing S3 buckets, VPCs).
 - Doesn't create anything, only **reads** data.

- ◆ Key difference:

resource = **creates** infrastructure
data = **reads** existing infrastructure

4. variable Block

Defines **input parameters** to customize configuration:

```
hcl
CopyEdit
variable "bucket_name" {
  type      = string
  description = "Used to set the bucket name"
  default    = "my-default-bucket"
}
```

Use in code via:

```
hcl
CopyEdit
bucket = var.bucket_name
```

- ◆ Benefit: Avoids hardcoding values.

5. **output** Block

Used to **expose values** after `terraform apply`:

```
hcl
CopyEdit
output "bucket_id" {
  value = aws_s3_bucket.my_bucket.id
}
```

- Helps extract IDs, names, ARNs, etc. from created resources.
 - Can also return values from variables or data sources.
-

6. **locals** Block

Defines **temporary internal variables** used for clarity or reusability:

```
hcl
CopyEdit
locals {
  local_example = "This is a local variable"
}
```

Use via:

```
h
CopyEdit
value = local.local_example
```

⚠ Syntax difference:

Use `local.<name>` (singular), not `locals.<name>`

7. **module** Block

Used to **reuse Terraform code** from external or local modules:

```
hcl
CopyEdit
module "mymodule" {
  source = "./module-example"
}
```

- `source` can be a:
 - Local folder
 - Git repo
 - Terraform Registry module
- Terraform loads that module's resources into your current config.

 Note: You need to actually have the `module-example/` folder or it will throw an error.

Summary of Terraform Blocks

Block	Purpose
<code>terrafo</code>	Project-level settings, required providers
<code>rm</code>	
<code>resourc</code>	Create/manage infrastructure
<code>e</code>	
<code>data</code>	Read existing infrastructure (external)
<code>variabl</code>	Accept input into Terraform configs
<code>e</code>	
<code>output</code>	Return values from Terraform to CLI/output
<code>locals</code>	Define reusable values within configuration
<code>module</code>	Reuse and import external Terraform code

Wrap-Up

- This lecture provided a **syntactic overview** of HCL (HashiCorp Configuration Language).
- We **did not run** Terraform — focus was only on **understanding how blocks are written**.
- Don't worry about memorizing every part — these blocks will reappear throughout the course.
- Refer to:
 - Terraform Language Docs
 - The **exercise_04** file if you want to practice writing these blocks

 **Next Up:** We'll start using these blocks more practically in actual infrastructure projects.

Let me know when you're ready for the next one.

◆ Lecture 4 Notes: Terraform Block, Version Constraints, Providers & Backends

Terraform Block Overview

- The `terraform` block configures **project-level settings**.
- Includes:
 - Required **Terraform version**
 - Required **provider versions**
 - **Backend configuration** (like S3 for remote state)
 - Optional: **cloud block** (for Terraform Cloud)

Example:

```
hcl
CopyEdit
terraform {
  required_version = "= 1.7.0"

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

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.33.0"
    }
  }
}
```

Input Variables Not Allowed

- **Only constants** allowed inside the `terraform` block.

- ❌ Cannot use `var.*` variables here (unlike in resources, modules, etc.).
-

Terraform Cloud

- `cloud` block configures **Terraform Cloud**, not discussed in detail here.
 - Deferred to a later lecture.
-

Backend Block

- Defines **where Terraform stores state files**.
 - In example: backend is S3.
 - Other backends supported (e.g., local, Consul, remote, etc.)
 - Fully explained later in the course.
-

Required Versions

- ♦ `required_version`
 - Refers to the **Terraform CLI version**.
 - Ensures Terraform used matches compatible version range.
 - ♦ `required_providers`
 - Defines **which provider** is used and its accepted version range.
 - Example uses `aws` with version `>= 5.33.0`.
-

1 2
3 4

Version Constraints Deep Dive

Types of Constraints

Symbol	Meaning
= 1.7.0	Exact match (only 1.7.0 allowed)
!= 1.6.2	Exclude version (disallow buggy release, e.g. 1.6.2)
>= 1.7.0	Version must be 1.7.0 or newer
<= 1.7.0	Version must be 1.7.0 or older
> 1.5.0	Any version above 1.5.0
< 2.0.0	Any version below 2.0.0
~> 5.33.0	Pessimistic constraint → allow 5.33.x only

`~>` means "lock major + minor, allow patch updates"

- `~> 5.33.0` → allows 5.33.1, 5.33.10, **but not** 5.34.0
- `~> 5.33` → same behavior
- `~> 1.5` → allows 1.5.1 to 1.5.x, **but not** 1.6.0

Practical Use Case

- If a provider version is buggy:
 - You can tell users to **exclude** that version (`!= x.y.z`)
 - Helps in avoiding known bad versions while keeping others open

Version Constraint Examples

hcl

CopyEdit

```
# Allow >= 1.7.0
required_version = ">= 1.7.0"

# Allow only 1.5.x (but not 1.6.0)
required_version = "~> 1.5.0"

# Range: allow 1.5.x but block 1.7+
required_version = ">= 1.5.0, < 1.7.0"

# Allow anything in 1.x series
required_version = "~> 1.0"
```

Key Takeaways

- **terraform block = project configuration center**
- Always **pin your Terraform and provider versions**
 - Ensures reproducibility and stability
- Use **~>** for **safe upgrades** (patches only)
- Don't use **var** in the **terraform** block

Lecture 5: First Complete Terraform Project with AWS S3 + Random Provider

Goal of This Lecture

- Create a **fully working Terraform project**.
- Use:
 - AWS Provider 
 - Random Provider 
- Deploy:
 - 1 x S3 Bucket with a random suffix.
- Learn the structure, syntax, and usage of:
 - `terraform` block (Terraform version)
 - `required_providers` block
 - Resource creation (`random_id`, `aws_s3_bucket`)
 - Output declaration (`output`)
 - Interpolation and referencing in HCL

Folder Structure & Setup

Create folder:

```
bash
CopyEdit
mkdir 03-first-terraform-project
cd 03-first-terraform-project
```

•

Create file:

```
bash
CopyEdit
touch providers.tf
```

-  File `providers.tf` will contain both:
 - `terraform` block (to define required version)
 - `required_providers` block (to define provider sources and versions)
 - AWS provider region setup
 -  Although some projects name this file `main.tf`, using `providers.tf` makes its purpose clearer (to manage providers).
-

Terraform Configuration File – `providers.tf`

```
hcl
CopyEdit
terraform {
  required_version = ">= 1.7.0"      # Ensures compatibility with
Terraform >= v1.7.0
  required_providers {
    aws = {
      source  = "hashicorp/aws"      # AWS provider from HashiCorp
      version = ">= 5.0"            # Ensures compatibility with AWS
    }
    random = {
      source  = "hashicorp/random" # Random provider for unique IDs
      version = ">= 3.0, < 4.0"     # Prevents breaking changes in v4
    }
  }
}
```

Why define `required_version`?

- Helps avoid compatibility issues if Terraform is run with an outdated version.
- Helps CI/CD environments auto-fail with the wrong Terraform version.

Why `random` provider?

- Used to generate unpredictable, collision-free names (e.g. S3 bucket names must be globally unique).
 - We'll append a random suffix to the bucket name.
-

Define AWS Region

Add to the same file:

```
hcl
CopyEdit
provider "aws" {
  region = "eu-west-1"  # Instructor's region. Replace with your
preferred AWS region.
}
```

 Replace with your own region if you're following along in a different zone (e.g., `us-east-1`, `ap-south-1`, etc.).

Resource 1: Generate a Random Suffix

```
hcl
CopyEdit
resource "random_id" "bucket_suffix" {
  byte_length = 6  # Generates 6 random bytes => 12 hex characters
}
```

 This resource will generate a **random ID** (hex format).

Useful because:

- S3 bucket names must be globally unique.
 - Using a suffix prevents collision even if multiple users use the same prefix.
-

Resource 2: Create an S3 Bucket

hcl

CopyEdit

```
resource "aws_s3_bucket" "example_bucket" {
  bucket = "example-bucket-${random_id.bucket_suffix.hex}" # Interpolated bucket name
}
```

 Breakdown:

- "`example-bucket-{$...}`" uses **string interpolation** to dynamically insert the random ID.
- `random_id.bucket_suffix.hex` accesses the **hexadecimal value** of the random ID generated.

 You could also use `.id`, `.b64`, or `.decimal` formats depending on your use case, but `.hex` is clean and readable.

Output the Bucket Name to Console

hcl

CopyEdit

```
output "bucket_name" {
  value = aws_s3_bucket.example_bucket.bucket
}
```

 This will print the generated bucket name **after** `terraform apply` is executed.

 This is especially useful for:

- Referencing in future modules
 - Debugging
 - User visibility during automation
-

CLI Commands to Run This Project

bash

CopyEdit

```
terraform init      # Initialize backend and providers
```

```
terraform plan      # See what Terraform will create
terraform apply    # Deploy the actual infrastructure
```

👀 After `apply`, you will see something like:

makefile

CopyEdit

Outputs:

```
bucket_name = "example-bucket-abc123def456"
```

🧠 Recap of What We Built

Component	Description
<code>terraform</code>	Defined Terraform CLI version compatibility
<code>required_providers</code>	Declared AWS + Random providers with version pinning
<code>provider "aws"</code>	Set AWS region
<code>random_id</code>	Generated a 12-character suffix using 6 random bytes
<code>aws_s3_bucket</code>	Created a unique-named S3 bucket
<code>output</code>	Displayed the final S3 bucket name after apply

🔗 Best Practices Demonstrated

- **✓ Version pinning** for providers avoids breaking changes.
 - **✓ Random ID** ensures **bucket name uniqueness**.
 - **✓ Clear file naming** (`providers.tf`) improves readability.
 - **✓ Use of interpolation** and **outputs**.
-

📁 Final File – `providers.tf`

hcl

```
CopyEdit
terraform {
  required_version = ">= 1.7.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.0"
    }
    random = {
      source  = "hashicorp/random"
      version = ">= 3.0, < 4.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1"
}

resource "random_id" "bucket_suffix" {
  byte_length = 6
}

resource "aws_s3_bucket" "example_bucket" {
  bucket = "example-bucket-${random_id.bucket_suffix.hex}"
}

output "bucket_name" {
  value = aws_s3_bucket.example_bucket.bucket
}
```

Lecture 6: Terraform CLI Deep Dive – Commands, Formatting, Planning, and Destruction

Objective

- Dive deeper into the Terraform CLI
 - Learn essential commands:
 - `init`, `fmt`, `validate`, `plan`, `apply`, `show`, `state list`, `destroy`, `help`
 - Understand proper file structure & execution flow
 - Understand how state and plans are managed
-

Refactoring: Better File Structure

- Previously, everything (providers + resources) was inside `providers.tf` – **bad practice**.
- Now, we:
 1. Keep `providers.tf` only for `terraform` and `provider` blocks.
 2. Create a new file `s3.tf` to hold S3 bucket and random ID resources.

 Terraform loads all `.tf` files in the **current directory** automatically – no imports needed.

Terraform's File Discovery Rules

Rule	Description
	Terraform reads all <code>.tf</code> files in the current directory
	Terraform ignores subdirectories unless explicitly handled (e.g., modules)

Command 1: `terraform init`

Initializes the working directory and downloads providers.

```
bash  
CopyEdit  
terraform init
```

 Downloads:

- `hashicorp/aws`
- `hashicorp/random`

 Re-run this command if:

- You add new providers
- You change provider versions
- You re-clone the repo or switch environments

AWS Credentials Setup

```
bash  
CopyEdit  
source .env
```

 Why?

- `.env` file holds AWS access credentials as environment variables
- Terraform automatically reads from environment if `~/aws/credentials` or variables like `AWS_ACCESS_KEY_ID` are set
- Avoids hardcoding credentials inside `.tf` files

Command 2: `terraform fmt`

```
bash
```

CopyEdit

```
terraform fmt  
terraform fmt -recursive
```

✓ Formats `.tf` files:

- Aligns `=` signs
- Indents blocks properly
- Makes HCL easier to read

Use `-recursive` to format all `.tf` files in the current directory **and all subdirectories**.

✓ **Command 3: `terraform validate`**

bash

CopyEdit

```
terraform validate
```

Checks for **syntactic correctness** of your configuration.

⚠ **Important Caveat:**

- Only validates structure & known argument types
 - It **won't** catch provider-specific or runtime errors like:
 - Invalid CIDR blocks
 - Invalid AMI IDs
 - AWS permissions issues
 - Always follow with a `terraform plan`
-

🧠 **Command 4: `terraform plan`**

bash

CopyEdit

`terraform plan`

Shows the **execution plan** – what changes Terraform will make.

 Green = create

 Yellow = modify

 Red = destroy

Saving a Plan File (Optional)

bash

CopyEdit

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

Creates a **binary file** of the plan that can be applied later.

- Not human-readable.
 - Used to **ensure consistency** (e.g., in CI/CD pipelines).
-

Command 5: `terraform apply`

bash

CopyEdit

```
terraform apply
```

- By default:
 - Shows plan
 - Asks for confirmation
- After confirming, executes the actions

Apply from Plan (No Prompt):

bash

CopyEdit

```
terraform apply myplan.tfplan
```

 No confirmation prompt. Be careful!

Command 6: `terraform show`

bash
CopyEdit
`terraform show`
`terraform show myplan.tfplan`

- Shows current **state file content** in a readable format.
- Can be used with `myplan.tfplan` to inspect saved plans.

 Shows *all details* returned by provider APIs, even if you didn't define them in your `.tf` file.

Command 7: `terraform state list`

bash
CopyEdit
`terraform state list`

- Lists resources tracked in the state.
- Cleaner and more concise than `terraform show`.

 Best for quick resource inventory:

CopyEdit
`random_id.bucket_suffix`
`aws_s3_bucket.example_bucket`

Command 8: Destroying Infrastructure

 Preview what will be destroyed:
bash
CopyEdit
`terraform plan -destroy`

Actually destroy:

```
bash
CopyEdit
terraform apply -destroy
```

Prompts for confirmation.

Without confirmation:

```
bash
CopyEdit
terraform apply -destroy -auto-approve
```

 `-auto-approve` can also be used with **regular apply** to skip confirmation.

Command 9: `terraform help`

```
bash
CopyEdit
terraform help
terraform apply -help
```

 Use to:

- Discover commands
 - Understand arguments & flags
 - Troubleshoot syntax
-

Final Clean-Up Tips

- Delete any leftover plan files:

```
bash
CopyEdit
rm myplan.tfplan
```

- Confirm nothing is left:

```
bash
CopyEdit
terraform state list
```

- Good habit: **destroy resources** after every lab to avoid AWS billing surprises.
-

✓ Summary of All Terraform CLI Commands

Command	Purpose
<code>terraform init</code>	Initializes Terraform and installs providers
<code>terraform fmt</code>	Auto-formats all Terraform files
<code>terraform validate</code>	Syntax validation of <code>.tf</code> files
<code>terraform plan</code>	Shows execution plan
<code>terraform plan -out=FILENAME</code>	Saves plan to binary file
<code>terraform apply</code>	Applies configuration with optional confirmation
<code>terraform apply FILENAME</code>	Applies a saved plan file
<code>terraform show</code>	Prints current state or a saved plan
<code>terraform state list</code>	Lists all managed resources
<code>terraform apply -destroy</code>	Destroys infrastructure
<code>terraform apply -destroy -auto-approve</code>	Destroys without confirmation
<code>terraform help</code>	Shows help and documentation

🧠 Best Practices Recap

- Use separate files for providers and resources (`providers.tf`, `s3.tf`)

- Use `terraform fmt` to auto-format before commits
- Always run `terraform plan` before `apply`
- Use `terraform state list` for cleaner insights than `show`
- Clean up with `terraform destroy + -auto-approve` after labs
- Use `-out` plans for safe, repeatable deployments in pipelines

Lecture: Terraform State (God-Level Notes)

♦ What is Terraform State?

- Terraform state is the **mechanism Terraform uses to map real-world infrastructure** to the definitions written in your Terraform `.tf` configuration files.
 - It serves as a **single source of truth** for what Terraform thinks is deployed.
-

What Does the Terraform State File Contain?

A typical **state file** (`terraform.tfstate`) contains:

Type	Purpose
Resource configurations	Records actual deployed resources that correspond to the <code>.tf</code> files
Bindings	Maps each Terraform block to real cloud resources (e.g., AWS EC2)
Metadata	Stores things like dependencies, lifecycle data, etc.
Backend configuration	If you're using a remote backend (like S3), the config details go here
Outputs	All defined <code>output</code> values are saved in state
Sensitive values	Credentials, secrets, tokens, etc., are stored here in plain text (⚠)

Example State File Snippet:

```
json
CopyEdit
{
  "version": 4,
  "terraform_version": "1.5.0",
  "resources": [
    {
      "type": "aws_instance",
      "name": "web",
```

```
"provider":  
"provider[\"registry.terraform.io/hashicorp/aws\"],  
    "instances": [  
        {  
            "attributes": {  
                "ami": "ami-0abc123456789def0",  
                "instance_type": "t2.micro",  
                "id": "i-0e12345678abc90de"  
            }  
        }  
    ]  
}  
}
```

Why Is the State File Sensitive?

- The state file **stores all outputs and variables**, including secrets like:
 - API keys
 - Access tokens
 - Passwords
-  **Security Warning:**
 - Never commit your `terraform.tfstate` file to version control (e.g., GitHub).
 - Protect it with strict access control (especially in teams).
 - Use encryption at rest if storing in remote backends.

State File: Required & Critical

- You **cannot use Terraform without a state file**.
- Every operation (plan, apply, destroy) reads from the state to determine:

- What resources exist
 - What needs to be created, updated, or deleted
-

How Terraform Uses the State File

1. Refresh:

- Before every `plan` or `apply`, Terraform **refreshes the state** by:
 - Querying cloud APIs (via providers)
 - Comparing what exists with what's in `.tf` files

2. Detect Drift:

- If something was changed **outside Terraform** (e.g., manually in AWS), and the `.tf` file wasn't updated:
 - Terraform calls this **configuration drift**
 - It **will try to revert** the infrastructure to match the `.tf` file

 Terraform always treats your `.tf` configuration as the source of truth.

Local vs Remote State

Storage Type	Description
Local (default)	State file stored on your local disk (<code>terraform.tfstate</code>)
Remote	Cloud-based storage like: <ul style="list-style-type: none">● Amazon S3 (AWS)● Google Cloud Storage (GCS)● Terraform Cloud

- Azure Blob Storage

|

👉 Remote backends enable **collaboration, locking, and versioning**.

🚫 State Locking

- When Terraform performs any **write operation** (like `apply`), it **locks the state file** to prevent multiple users from making conflicting changes.

🔄 Scenario:

- **Dev A** is applying infra.
- **Dev B** tries to run `terraform apply`.
- ⚠️ Dev B will see a lock error and must wait for Dev A to finish.

🔒 This prevents **state corruption** and ensures consistency.

🧠 Recap – Key Concepts

Concept	Summary
Terraform state	Maps <code>.tf</code> to real-world infrastructure
Required	Yes – Terraform won't function without it
Sensitive data	Contains secrets – secure it carefully
Prevents drift	Makes real infra match <code>.tf</code> files
Storage options	Local (default) or remote (e.g., S3)
State locking	Prevents concurrent changes
Configuration is truth	<code>.tf</code> files are always considered authoritative

💡 Best Practices

✓ DO:

- Store state in a **remote backend** (e.g., S3 with versioning and encryption).
- **Use locking** (enabled by default in remote backends like S3 + DynamoDB).
- Limit access via IAM policies or secrets management tools.
- Use `.terraformignore` or `.gitignore` to avoid checking state files into Git.

DON'T:

- Don't edit the state file manually.
- Don't let multiple users modify infrastructure without locking.
- Don't leave the state file unsecured or unencrypted.

Coming Up Next

- How to configure **remote backends** (e.g., S3 + DynamoDB locking)
- Terraform Cloud usage
- Hands-on examples of state drift, recovery, and remote collaboration

Lecture: Terraform State – Local Backend, Inspecting `.tfstate`, Lock Files, Backups

1. Recap and Context

- Continuing the **Data Form Project** using Terraform.
 - This lecture focuses on understanding **Terraform State**—what it is, how it's stored, what it contains, and how Terraform uses it.
 - We are working in a **local environment**, not Terraform Cloud or any remote backend.
-

2. Changing into the Project Directory

```
bash
CopyEdit
cd <your-project-directory>
```

- Move into the directory where Terraform configuration (`.tf`) files are stored.
-

3. Initializing Terraform (`terraform init`)

```
bash
CopyEdit
terraform init
```

- Initializes the project.
- Downloads necessary **providers** defined in `.tf` files.
- Since this is a previously used project, **same provider versions** are reused.
- Creates the hidden folder:

```
bash
CopyEdit
```

.terraform/

- Inside .terraform, you'll see:
 - The **AWS Provider plugin**
 - The **random provider plugin**
-

📁 4. Terraform State: Local vs Remote

- ! Since there's **no backend block** in our Terraform configuration, it **defaults to local state**.
- Local state means:

State data is stored in a local file:

CopyEdit

`terraform.tfstate`

- - No collaboration support (used only by a single machine/user).
-

🔍 5. Planning Infrastructure (`terraform plan`)

bash

CopyEdit

`terraform plan`

- This command:
 - Creates **temporary plan files** (e.g., `.terraform.tfplan`) to compare actual infra vs desired state.
 - These files are **deleted automatically** after the command finishes unless explicitly saved.
-

6. Applying the Plan (`terraform apply`)

```
bash
CopyEdit
terraform apply
```

- Runs the actual provisioning of resources.
- User manually approves (`yes`) when prompted.
- After this, Terraform creates:

```
bash
CopyEdit
terraform.tfstate
```

- This is the **permanent state file** and will be updated on every change.
- It reflects the current **real infrastructure state**.

7. Inspecting `terraform.tfstate`

- Format: **JSON**
- Contains:
 - **Outputs** block (values defined in `output.tf`)
 - **Resources array** (each managed resource):
 - For each resource:
 - Type (`aws_s3_bucket`, `random_id`, etc.)
 - Provider used
 - `id`, `arn`, bucket name, domain, etc.
 - Metadata
 - **Dependencies array**:

- Lists IDs of resources this depends on (important for **resource creation order**)

 *Example:*

```
json
CopyEdit
{
  "resources": [
    {
      "type": "aws_s3_bucket",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "attributes": {
            "id": "my-bucket-name",
            "arn": "arn:aws:s3:::my-bucket-name",
            ...
          },
          "depends_on": [ "random_id.bucket_suffix" ]
        }
      ],
      {
        "type": "random_id",
        "instances": [
          {
            "attributes": {
              "id": "abc123"
            }
          }
        ]
      }
    ]
  }
}
```

 **Note:**

- Even a simple 2-resource project can produce **100+ lines** in the state file.
 - In real projects, `.tfstate` files can have **thousands of lines** and sensitive data (e.g., keys, passwords).
-

8. State File Security and Best Practice

- Don't manually edit `.tfstate`.
- Never commit `.tfstate` to Git.
- Use `.gitignore`:

CopyEdit

```
terraform.tfstate
terraform.tfstate.backup
.terraform/
```

9. Destroying Infrastructure (`terraform destroy`)

bash

CopyEdit

```
terraform destroy
```

- When you run it:

A **lock file** is created:

pgsql

CopyEdit

```
terraform.tfstate.lock.info
```

○

- This is **state locking** to prevent simultaneous operations (important in team workflows).

 Lock File Contents:

json

CopyEdit

```
{
```

```
  "ID": "some-id",
  "Operation": "destroy",
  "Created": "timestamp",
  "Info": {
    "TerraformVersion": "1.x.x",
```

```
    ...
}
}
```

📌 While Terraform is waiting for user confirmation ([yes](#)/[no](#)), the lock file exists.

- If you answer [no](#), the file disappears (lock released).
 - If you answer [yes](#), the destruction starts and finishes, and then the lock is removed.
-

10. Terraform Backup File

- After destroying infra, this file is created:

CopyEdit

`terraform.tfstate.backup`

- It stores the **previous state** before the destruction.
- Acts as a fallback in case destruction fails.

 Contents:

- Exact copy of `.tfstate` before destroy operation.
 - Used rarely in practice.
 - More common in **remote state backends** (e.g., S3 with versioning, Terraform Cloud).
-

Best Practice: Remote Backend

- For collaboration & resilience:
 - Use S3 backend with **versioning + DynamoDB locking**

hcl

CopyEdit

```
terraform {
  backend "s3" {
    bucket      = "my-terraform-state"
    key         = "dev/terraform.tfstate"
    region      = "us-west-2"
    dynamodb_table = "terraform-locks"
  }
}
```

Summary of Important Files

File	Description
<code>terraform.tfstate</code>	Main state file (current infra)
<code>terraform.tfstate.backup</code>	Backup of previous state
<code>.terraform/</code>	Hidden folder containing providers
<code>.terraform.lock.hcl</code>	Provider version locking
<code>terraform.tfstate.lock.info</code>	Temporary lock file during operations

Commands Recap

bash
CopyEdit

```
terraform init      # Initialize providers
terraform plan      # Preview changes
terraform apply      # Apply infrastructure
terraform destroy    # Tear down resources
```

Sensitive Info Warning

- `.tfstate` often contains:
 - Secrets

- Keys
- Credentials
- Never share or publish state files.



Terraform Backends – GOD-LEVEL NOTES

◆ What is a Backend in Terraform?

- A **backend** defines where Terraform stores its state file.
- The **state file** (`terraform.tfstate`) is the **source of truth** for Terraform — all infrastructure metadata lives here.
- **Proper storage and access control** are **critical** for:
 - Consistency
 - Collaboration
 - Security
 - Disaster recovery

Types of Terraform Backends

There are **3 broad categories**:

① Local Backend

- **Default behavior** when no backend block is specified.
- The state file is stored **on the same machine** (typically in the current project folder).
- Not recommended for team or production use.

Example path:

```
bash
CopyEdit
./terraform.tfstate
```

•

2 Terraform Cloud Backend

- Terraform's own hosted SaaS backend.
- Not **just** for state storage:
 - Includes **state locking, version history, remote runs, teams, VCS integrations, workspaces**.
- Requires:
 - Terraform Cloud **account**
 - API tokens or CLI auth
- State is stored in **Terraform Cloud Projects/Workspaces**
- More coverage later in the course.

3 Third-Party Remote Backends

Examples:

- AWS S3
- Google Cloud Storage
- Azure Blob Storage
- Consul
- etcd
- HTTP(s)

 These are called "remote" because the state file is stored **away from the local machine**.

Differences Between Backends

Feature	Local	Remote (S3, GCS, etc.)	Terraform Cloud
Team collaboration	✗	✓ (w/ proper config)	✓✓✓
State locking	✗	✓* (only for some backends like S3 with DynamoDB)	✓✓✓
Versioning	✗	✓ (if backend supports)	✓✓✓
Extra features (VCS, UI, Runs)	✗	✗	✓✓✓

⚠️ Not all remote backends support **state locking** or **versioning**.

🔗 Backend Block Rules

When configuring a backend in Terraform, remember:

- Only **one backend can be defined** per project.
 - **Cannot merge multiple backends.**

Backend block is defined inside the **Terraform block**:

```
hcl
CopyEdit
terraform {
  backend "s3" {
    bucket = "my-tf-state"
    key    = "prod/terraform.tfstate"
    region = "us-west-2"
  }
}
```

-
-

⚠️ Limitations of Backend Block

- Cannot use:

- **Variables** (no `var.` references)
 - **Resources** (no `aws_` or similar)
 - **Data sources**
 - Must be **static** configuration.
-

Authentication for Remote Backends

- Terraform must **authenticate** to access the backend.
 - Example (S3):
 - Requires:
 - `AWS_ACCESS_KEY_ID`
 - `AWS_SECRET_ACCESS_KEY`
 - Permissions required:
 - **Read**: To fetch the current state
 - **Write**: To save updates to the state
 - **Delete**: Optional, for cleanup
 - **Read-only access is not sufficient**
-

Migrating Between Backends

Changing a backend requires:

`csharp`
`CopyEdit`
`terraform init`

-

- Backend migration **is supported**, but:
 - **Not always smooth**
 - Sometimes manual migration is safer:
 - Export old state file
 - Upload it to new backend manually
 - `terraform init` with the `-migrate-state` flag helps auto-migrate.
-



Best Practice

- Set the backend **once**, early in the project.
 - Avoid frequent changes — it's a **foundational configuration**.
 - If needed, migrate carefully.
-



Summary Table

Backend Type	Storage Location	Extra Features
Local	Local disk	✗
Remote (S3, etc.)	Remote storage (S3, GCS)	✓ Some (lock/version)
Terraform Cloud	Terraform's SaaS backend	✓✓✓ (Full platform)



Summary

- Backends determine **how and where** Terraform stores state.
- Types:

- **Local** – simple, but not collaborative.
 - **Terraform Cloud** – advanced features for teams.
 - **Remote (S3, GCS, etc.)** – good for self-managed teams.
- Configure **only one backend** per project.
 - Must re-run `terraform init` after backend changes.
 - State **must be protected**: use remote backends with proper credentials and access control.

Lecture Notes: Terraform Backends – Deep Dive into S3 Backend Configuration

What is a Backend in Terraform?

A **Terraform backend** defines **where and how Terraform stores the state file**.

This `.tfstate` file is the **source of truth** for managing infrastructure, and thus **must be securely stored** and properly accessed.

If this state file is:

- **Deleted or lost** → Terraform loses all knowledge of the real-world infrastructure.
 - **Corrupted or inconsistent** → It can cause dangerous misconfigurations or resource destruction.
-

Types of Backends

Terraform supports **three categories of backends**:

1. Local Backend

- Default.
- Stores `terraform.tfstate` **locally on your machine**.
- Not safe for team/shared projects.

Example:

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

}

-
-

2. Terraform Cloud Backend

- Hosted by HashiCorp.
- Not just a storage backend, but includes:
 - Remote execution
 - Workspaces
 - Collaboration features
- Stores state **inside a "Terraform Cloud Workspace"**.

Example (simplified):

```
hcl
CopyEdit
terraform {
  backend "remote" {
    organization = "my-org"

    workspaces {
      name = "my-workspace"
    }
  }
}
```

-
-

3. Remote Backends (3rd Party)

- Uses external services like:
 - AWS S3
 - Azure Blob Storage

- Google Cloud Storage (GCS)
 - **Most common for real-world teams and automation.**
 - Example (S3 backend shown below).
-



Backend Comparison Table

Backend Type	Example	State Locking Support	Multi-user Safe	Extra Features
Local	Local disk	✗	✗	✗
Terraform Cloud	HashiCorp	✓	✓	✓ (UI, VCS etc)
Remote (e.g., S3)	AWS S3	✓ (with DynamoDB)	✓	✗



S3 Backend Configuration – Practical Setup

To use an S3 backend in Terraform:

1. Configure in your `main.tf`:

```
hcl
CopyEdit
terraform {
  backend "s3" {
    bucket      = "my-terraform-state-bucket"
    key         = "env/dev/terraform.tfstate"
    region      = "us-east-1"
    encrypt     = true
    dynamodb_table = "terraform-locks"
  }
}
```



Explanation:

Field	Meaning
-------	---------

<code>bucket</code>	Name of the S3 bucket that holds your state file
<code>key</code>	Path within the bucket (acts like a file name)
<code>region</code>	AWS region of the S3 bucket
<code>encrypt</code>	Enables encryption-at-rest via S3-managed keys
<code>dynamodb_table</code>	Optional. Required if you want state locking via DynamoDB

Authentication for Remote Backends

When using remote backends like S3, Terraform must authenticate:

-  **Required:** IAM credentials with **read/write access** to:
 - The S3 bucket (`GetObject`, `PutObject`, `ListBucket`)
 - The DynamoDB table (`GetItem`, `PutItem`, `DeleteItem`, `UpdateItem`) — if used

 **Important: Read-only access is not enough** — Terraform updates the `.tfstate` file continuously.

Backend Block Limitations

The `backend` block (inside `terraform {}`) has **strict rules**:

Limitation	Explanation
 No Input Variables	You cannot use variables like <code>var.bucket_name</code> inside a backend block
 No Resource/Data References	You can't use <code>\$(aws_s3_bucket.mybucket.bucket)</code> inside it
 Hardcoded only	All values must be static strings or hardcoded values

This is because the backend needs to be initialized **before** any Terraform resource is evaluated.

Changing or Migrating Backends

Changing a backend (e.g., local → S3) **requires special handling**:

Terraform offers a built-in migration tool:

```
bash
CopyEdit
terraform init
```

- When it detects a **backend change**, it will prompt:

"Do you want to copy the existing state to the new backend?"

- **Works most of the time**, but not always reliable. In some edge cases, manual migration may be needed:
 - `terraform state pull` → download existing state
 - `terraform state push` → push it to new backend

Important Rules & Tips

Rule/Fact	Why it matters
 You can configure only one backend per Terraform project	No multi-backend support. You can't merge two backends.
 Any time you change backend config , you must rerun <code>terraform init</code>	This reinitializes Terraform to work with the new backend.
 Not all backends support state locking	For S3, locking requires DynamoDB setup. Others may not support it at all.
 No backend = no locking, no collaboration safety	Local state is unsafe for teams or production.
 Set up the backend early in the project	Don't delay backend config. Migrating state later is risky and error-prone.



Visual Summary – Backend Types

sql

CopyEdit

Local Backend
local .tfstate

Terraform Cloud
Remote .tfstate
UI, VCS, etc.

Remote Backend
(e.g., AWS S3)
S3 + DynamoDB



Final Takeaways

- Terraform backends **define state storage** and **collaboration ability**.
- Use **S3 + DynamoDB** in production for safe, concurrent access.
- Backend configs are **not flexible** — no variables, no resources inside them.
- Use `terraform init` to initialize or migrate backends.
- Do it **early**, avoid manual state migration later.

Lecture Notes – Partial Backend Configuration in Terraform (Dev vs Prod)

Objective

We aim to configure **different backends** (e.g., dev and prod) using **partial configurations**, and explore different ways to supply backend settings:

- Provide backend config via separate files
 - Supply partial backend config in `.tf` files and complete it via CLI
 - Understand how Terraform behaves when switching/migrating state
 - Practice usage patterns common in **CI/CD environments**
-

What is Partial Backend Configuration?

Instead of fully defining the `backend` block **inside** your `main.tf`, you can:

- Leave it **completely empty**
- Provide the values externally via CLI flags or external files

This is useful for **environment separation** (dev/prod/staging) or **CI/CD pipelines**.

Step-by-Step Implementation

Step 1: Move Backend Block to a Separate File

 Previously inside `main.tf`:

```
hcl
CopyEdit
terraform {
  backend "s3" {
```

```
        bucket      = "my-bucket"
        key        = "04-backend/state.tfstate"
        region     = "eu-west-1"
        encrypt    = true
        dynamodb_table = "terraform-locks"
    }
}


```

Step 2: Create a new backend config file

Create: `dev.s3.tfbackend`

File content:

```
hcl
CopyEdit
bucket      = "my-bucket"
key        = "04-backend/dev/state.tfstate"
region     = "eu-west-1"
encrypt    = true
dynamodb_table = "terraform-locks"
```

Note: File name convention → `<env>. <backend>.tfbackend`

Step 3: Initialize Terraform with External Backend Config

```
bash
CopyEdit
terraform init \
  -backend-config="dev.s3.tfbackend" \
  -migrate-state
```

- `-backend-config` points to the file containing backend values
 - `-migrate-state` moves existing state to new location (dev folder)
-

Step 4: Verify and Use Plan Files

```
bash
CopyEdit
terraform plan -out=dev_plan
terraform apply dev_plan
```

- Saves the plan to a binary file `dev_plan`
 - Executes the plan without confirmation prompt
-

Step 5: Clean Up Old State File in S3

Navigate to S3 console:

- Go to bucket → `04-backend/`
 - Delete the old state file that was migrated
 -  **Note:** S3 doesn't use folders; the UI just displays key prefixes like folders.
-

Create Another Environment (Production)

Copy Dev File → Create: `prod.s3.tfbackend`

```
hcl
CopyEdit
bucket      = "my-bucket"
key         = "04-backend/prod/state.tfstate"
region       = "eu-west-1"
encrypt      = true
dynamodb_table = "terraform-locks"
```

Init for Prod

```
bash
CopyEdit
terraform init \
  -backend-config="prod.s3.tfbackend" \
  -migrate-state
```

✓ Now the state is moved to the `prod` subpath.



Why Avoid Switching Backends Locally?

If you're switching between `dev` and `prod` in your **local environment**, the reinitialization every time is **tedious and risky**.

Instead:

- Use **CI/CD pipelines** where `terraform init` is called on each run
 - Provide correct backend config depending on environment
-



Alternative: Partial Backend Config via CLI (No File)

Example

Partial config inside `.tf` file:

```
hcl
CopyEdit
terraform {
  backend "s3" {
    bucket      = "my-bucket"
    key         = "04-backend/state.tfstate"
    encrypt     = true
    dynamodb_table = "terraform-locks"
    # region not provided here
  }
}
```

Then CLI:

```
bash
CopyEdit
terraform init \
  -backend-config="region=eu-west-1" \
```

```
-migrate-state
```

Terraform merges:

- Hardcoded values from `.tf` file
- Values from CLI

 Works just like full config.

Final Cleanup Steps

Add Comments in Unused Files

Mark old backend config files as unused:

```
hcl
CopyEdit
# dev.s3.tfbackend
# For illustration only – not actively used
```

Revert to Single Backend Config (Full config again)

Restore full backend block in `main.tf`:

```
hcl
CopyEdit
terraform {
  backend "s3" {
    bucket      = "my-bucket"
    key         = "04-backend/state.tfstate"
    region      = "eu-west-1"
    encrypt     = true
    dynamodb_table = "terraform-locks"
  }
}
```

Then re-init:

```
bash
```

CopyEdit

```
terraform init -migrate-state
```



Destroy Everything (Optional Final Step)

bash

CopyEdit

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

Ensures no active resources or state left behind

After destroy: state file shows **no resources** – which is intended



Summary: Partial Backend Config Patterns

Method	Description
.tf with full backend block	Most common and static
.tf with partial backend block + CLI	Offers dynamic overrides via CLI
.tf with empty backend block + *.tfbackend file	Externalized config ideal for CI/CD pipelines



Final S3 Structure (after cleanup)

arduino

CopyEdit

04-backend/

```
|   └── state.tfstate   Current config
```

Deleted:

- dev/state.tfstate
- prod/state.tfstate



What Are Providers in Terraform?

- **Providers** are **plugins** that allow Terraform to **interact with remote APIs**, e.g.:
 - AWS
 - GCP
 - Azure
 - Kubernetes
 - GitHub
 - Custom REST APIs
 - **Terraform Core** cannot manage resources directly — it delegates to providers.
-



Declaring Providers in Terraform

hcl
CopyEdit

```
terraform {  
    required_version = ">= 1.0.0"  
  
    required_providers {  
        aws = {  
            source  = "hashicorp/aws"  
            version = ">= 5.0.0, < 5.5.0"  
        }  
    }  
}
```

- **source**: Identifies where to download provider from
- **version**: Use version constraints (`>=`, `<`, `~>`, etc.)

Best practice: Match the **provider key** (`aws`) in `required_providers` with the **provider block name** and the **prefix of resource types** (like `aws_s3_bucket`)

Notes on Provider Behavior

- Each provider introduces **resource types** and **data sources**
- You **must include** all required providers in the root module.
- Child modules will **inherit** provider configs unless overridden.
- Dependency lock files (`.terraform.lock.hcl`) **should be committed** to Git.
 - They lock in specific plugin versions

Demo: Provider Configuration

Folder: 05-providers

File: `providers.tf`

Basic Setup

hcl
CopyEdit

```
terraform {  
    required_version = "~> 1.0.0"  
  
    required_providers {  
        aws = {  
            source  = "hashicorp/aws"  
            version = ">= 5.0.0, < 5.5.0"  
        }  
    }  
}  
  
provider "aws" {  
    region = "eu-west-1"  
}
```

Create an S3 Bucket

hcl

CopyEdit

```
resource "aws_s3_bucket" "my_bucket" {
  bucket = "some-random-bucket-420abcd" # make it unique!
}
```

⚠ Experiment: Renaming Provider (Don't Do This)

You **could** do this:

```
hcl
CopyEdit
required_providers {
  whatever = {
    source  = "hashicorp/aws"
    version = ">= 5.0.0"
  }
}

provider "whatever" {
  region = "eu-west-1"
}
```

But this causes issues:

- Terraform uses the **prefix** of a resource (`aws_s3_bucket`) to **match the provider name** (`aws`)
- You get this warning:

pgsql

CopyEdit

```
Error: Provider configuration not found
```

```
  Required provider aws was not declared in required_providers
```

✓ **Conclusion:** Always **match provider key** with **resource prefix** to avoid confusion.

🌐 Multi-Region Deployments

You can deploy to **multiple regions** using **provider aliases**.

Define Multiple Provider Instances

```
hcl
CopyEdit
provider "aws" {
  region = "eu-west-1"  # default
}

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

Use the Aliased Provider

```
hcl
CopyEdit
resource "aws_s3_bucket" "eu_west_bucket" {
  bucket    = "some-eu-west-unique-123"
}

resource "aws_s3_bucket" "us_east_bucket" {
  provider = aws.us_east
  bucket   = "some-us-east-unique-456"
}
```

 Resources use **default provider** unless `provider = aws.alias_name` is set explicitly.

Commands + Output

Initialize Project

```
bash
CopyEdit
terraform init
```

Installs providers, locks versions in `.terraform.lock.hcl`

Plan and Apply

```
bash
CopyEdit
terraform plan
terraform apply -auto-approve
```

→ Two S3 buckets created, one in **Ireland**, one in **North Virginia**

Destroy

```
bash
CopyEdit
terraform destroy -auto-approve
```

→ Deletes both S3 buckets

Changing Provider Versions

Change Version Constraint in `providers.tf`

```
hcl
CopyEdit
# Example downgrade:
version = "< 5.0.0"
```

Update Installed Version

```
bash
CopyEdit
terraform init -upgrade
```

→ Reinstalls provider version that fits new constraint

Check Installed Versions

```
bash
CopyEdit
terraform version
```

Shows:

- Terraform version
 - Installed provider versions (actual, not declared)
-

Does Changing Provider Version Affect State?

-  No — changing version **does not affect** the `terraform.tfstate`
 -  Unless:
 - Major version introduces **breaking changes**
 - You change **resource arguments** as a result
-

Summary / Key Takeaways

- Providers are plugins used by Terraform to talk to external systems
- Declare providers in `required_providers` block with proper `source` and `version`
- Use **aliases** to configure **multi-region setups**
- Never rename providers unless you know how to wire them properly
- Use `terraform version` and `.terraform.lock.hcl` to manage provider consistency
- Changing provider version → run `terraform init -upgrade`

Resources

Section: Terraform Resources - Lecture 1: Introduction to Resources

What is the purpose of this section?

This section introduces you to the **core building block of Terraform – the `resource` block**.

It explains:

- What a resource is
- Why it's central to Terraform
- How resources translate to real-world infrastructure (e.g., EC2, S3)
- How arguments differ per resource type
- Rules about resource uniqueness
- AWS-specific behavior (like tags)
- Local-only (non-cloud) resources like `random_id`
- Future possibilities: looping to create multiple resources

Concept: What are Resources in Terraform?

Resources represent **real-world infrastructure components** you want to manage using Terraform.

Think of them as:

- EC2 instances
- S3 buckets
- DNS records
- Firewalls

- IAM roles
- Databases
- Kubernetes pods
- TLS certs
- Virtual networks
- Load balancers

Every cloud service and infrastructure provider (AWS, GCP, Azure, etc.) exposes **hundreds to thousands** of resource types.

Anatomy of a Resource Block

Here's what a Terraform `resource` block typically looks like:

```
hcl
CopyEdit
resource "<PROVIDER>_<RESOURCE_TYPE>" "<RESOURCE_NAME>" {
    # arguments specific to the resource
}
```

For example:

```
hcl
CopyEdit
resource "aws_instance" "example" {
    ami          = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}
```

Key Components:

- `aws_instance` → Provider + resource type
- `example` → Logical name (local to the module)

- **Inside the block** → Arguments (e.g., `ami`, `instance_type`)
-

Key Concepts Covered in Lecture

1. Resources Are Central to Terraform

- 90%+ of any Terraform project is made of resource blocks.
 - They represent actual infrastructure.
 - Everything else (variables, outputs, locals, etc.) supports resource creation.
-

2. Resource Arguments Vary by Type

Each resource has its own set of valid arguments.

Resource Type	Valid Argument Example
---------------	------------------------

```
aws_instance ami, instance_type,  
           tags  
  
aws_s3_bucket bucket, acl,  
                versioning  
  
random_id     byte_length
```

If you pass an invalid argument (e.g., `bucket` to `aws_instance`), **Terraform will throw an error** at plan/apply time.

 Built-in validation ensures correctness.

3. Tagging Behavior is Provider-Specific

- **AWS** supports `tags` block for many resources:

```
tags = {
  Name = "MyInstance"
  Env  = "Dev"
}
```

- But not all AWS resources support tags.
- **Other providers** (e.g., GCP, Azure) might **use different tagging mechanisms**.
 - GCP might use **labels**, not **tags**.
 - Azure might have **tags**, but structure may vary.

📌 You must always check the official Terraform documentation per provider.



Use:

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/>

4. **Resource Type + Name Must Be Unique Within a Module**

hcl

CopyEdit

```
resource "aws_instance" "web" {}
resource "aws_instance" "web" {} # ✗ Error: duplicate name
```

Why?

Terraform internally uses the combo of `resource_type.resource_name` to **track and bind the resource to real-world infra**.

It helps Terraform:

- Compare planned vs actual infrastructure
 - Track changes
 - Maintain state file consistency
-

5. **Default vs Custom Providers**

By default, Terraform uses the **default provider** configured in your `provider` block.

Example:

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

But you can override it per resource using:

```
hcl
CopyEdit
resource "aws_instance" "example" {
  provider = aws.europe
}
```

This is useful when you're managing resources in **multiple regions/accounts**.

6. Some Resources Are Local-Only (No Cloud Infra Created)

Example:

```
hcl
CopyEdit
resource "random_id" "server" {
  byte_length = 8
}
```

This doesn't create any AWS resource. It just generates a **random ID** that you can use as part of a name.

Other local-only examples:

- `tls_private_key` → Generate private key
- `tls_self_signed_cert` → Generate self-signed TLS cert
- `local_file` → Write a file locally

 These are **tools to support infra deployment**, not infra themselves.

7. Creating Multiple Resources with Loops

You can use loops like:

- `count` (create N instances)
- `for_each` (create from map or list)

Example (not covered yet but previewed):

```
hcl
CopyEdit
resource "aws_instance" "server" {
    count      = 3
    ami        = "ami-xyz"
    instance_type = "t2.micro"
}
```

Each instance will be:

- `aws_instance.server[0]`
- `aws_instance.server[1]`
- `aws_instance.server[2]`

 Full explanation in future lectures.

Summary – Key Takeaways

Concept	Meaning
Resource Block	Core building block to define infrastructure
Arguments Vary Per Resource	e.g., <code>ami</code> is valid for <code>aws_instance</code> but not for <code>aws_s3_bucket</code>
Tags Are AWS-Specific	May not work in other providers or all AWS resources

Resource Type + Name Must Be Unique	Helps Terraform track real-world resource bindings
Default Provider Used Unless Overridden	Can use multiple providers using <code>provider</code> argument
Local-only Resources Exist	e.g., <code>random_id</code> , <code>tls_private_key</code> , <code>local_file</code>
Loops Create Multiple Resources	<code>count</code> / <code>for_each</code> allows scaling resources easily



Terraform Official Docs (Suggested Search Terms)

- `resource` block syntax
 - `aws_instance` arguments
 - `random_id`
 - `provider` argument in resource block
 - `tags` for AWS resources
 - `count` and `for_each`
-



Next Steps

In the **next lecture**, we will:

- Actually **write resource blocks**
- Deploy real infrastructure
- See how Terraform initializes, plans, and applies changes

Lecture 2: Terraform Resource Dependencies

— complete, deeply explained, with **no detail skipped**:

Goal of the Lecture

Understand how **Terraform handles resource creation order**, both **implicitly** and **explicitly**, using dependencies between resources.

This is **critical in real-world infrastructure** where resources like EC2, VPCs, IAM roles, etc. have interdependencies — and wrong creation order leads to failure.

Core Concept: Dependency Management in Terraform

Terraform's job is to:

- Figure out **which resources are independent** and can be created **in parallel**.
- Figure out **which resources depend on others** and must be created **in sequence**.

It uses:

- **Expressions inside resource blocks** to detect **implicit dependencies**
 - Manual hints using `depends_on` to declare **explicit dependencies**
-

How Terraform Determines Dependencies (with Examples)

Example Resources:

- `aws_vpc.main`

- `aws_s3_bucket.logs`
 - `aws_iam_role.ec2_role`
 - `aws_instance.web_server`
 - `aws_db_instance.mysql`
-

✓ Implicit Dependencies (Terraform figures them out automatically)

Let's say:

- EC2 Instance refers to a **subnet ID** → which comes from a **VPC**
- RDS refers to a **subnet group** → which also comes from a **VPC**

So both **EC2** and **RDS** indirectly **reference the VPC**, meaning:

```
plaintext
CopyEdit
aws_vpc.main → aws_instance.web_server
aws_vpc.main → aws_db_instance.mysql
```

Terraform creates:

- **VPC first**
- Then **EC2 and RDS in parallel** (because they're both dependent on the same VPC, but **not on each other**)

 **Why parallel?** Because EC2 and RDS don't reference each other — only the common VPC.

🚫 No Dependencies = Parallel Creation

Let's say:

- IAM role is only used by EC2

- IAM role doesn't reference anything inside the VPC or S3

Then Terraform will **create the IAM role separately and early**, in **parallel** with VPC or S3 bucket.

Conditional Dependencies

If the **IAM role starts referencing an S3 bucket** (e.g., allows access to that bucket), then:

Terraform recognizes this new dependency:

```
plaintext
CopyEdit
aws_s3_bucket.logs → aws_iam_role.ec2_role
```

Now it will ensure the **S3 bucket is created before the IAM role**.

Explicit Dependencies Using `depends_on`

Sometimes Terraform **can't figure out the dependency** (e.g., no variable reference, indirect relationship).

You can manually tell Terraform the dependency:

```
hcl
CopyEdit
resource "aws_iam_role" "ec2_role" {
  name = "my-role"

  depends_on = [
    aws_s3_bucket.logs
  ]
}
```

This ensures that the **S3 bucket is created first**, even if there is no direct reference.

If a Parent Resource Fails?

If an upstream resource like `aws_vpc.main` **fails to create**, then:

Terraform **won't even try** creating EC2 or RDS because they **depend on the VPC**.

This avoids:

- Unnecessary API calls
 - Invalid errors and confusion
-

`replace_triggered_by` Meta Argument

Rarely Used But Important

It allows you to **force replacement of a parent resource** if a child resource changes.

Example use case:

- You want to **replace the IAM role** if the **EC2 instance** is updated
- Add `replace_triggered_by` in the IAM role:

```
hcl
CopyEdit
resource "aws_iam_role" "ec2_role" {
  name = "my-role"

  lifecycle {
    replace_triggered_by = [
      aws_instance.web_server
    ]
  }
}
```

 Real world use is rare but powerful in edge cases.

Summary of Behavior

Situation	Terraform Behavior
-----------	--------------------

Resources have no references	Created in parallel
Resources reference each other	Created in sequential order
Dependency not obvious to Terraform	Use <code>depends_on</code>
Upstream resource fails	Downstream not executed
Force replacement of one by another	Use <code>replace_triggered_by</code>

Visual Concept Recap (from Lecture Diagram)

- VPC is the **root dependency**
 - S3 Bucket is **independent**
 - IAM Role (if isolated) is **independent**
 - RDS and EC2 both **depend on VPC**, created **after VPC but in parallel**
-

Terraform Docs for Reference

- `depends_on`
 - `replace_triggered_by`
 - Terraform Resource Lifecycle
 - Terraform Graph Theory (Advanced)
-

Next Steps After Theory

The instructor concludes the theory here and prepares for hands-on practice with actual dependency declarations and resource creation.

Lecture 3 – Meta-Arguments in Terraform (God-Level Notes)

What are Meta-Arguments?

- Meta-arguments are **special configuration options** in Terraform that **modify how a resource behaves** rather than what it does.
- They **do not define infrastructure**, but control **how Terraform creates, updates, or destroys** resources.

 Think of meta-arguments as "**Terraform's behavioral switches**" — toggles that change **when, how many, under what dependencies, and under what conditions** a resource is managed.

 **Official Terraform Doc:**
Meta-Arguments - Terraform

1. depends_on

Purpose:

To **manually enforce dependency** between resources — telling Terraform:

“Wait for this before doing that.”

Syntax:

hcl

CopyEdit

```
resource "aws_instance" "web" {  
    ami          = "ami-xyz"  
    instance_type = "t2.micro"  
    depends_on    = [aws_s3_bucket.logs]  
}
```

Why it's needed:

Terraform automatically detects dependencies when a resource **references** another's attribute.

But **if there's no direct reference**, Terraform might incorrectly **parallelize** their creation.

👉 Use `depends_on` when:

- There's an **implicit dependency**.
- Resources are created in **parallel but must be sequential**.
- You're using **provisioners**, `local-exec`, or side-effects.

🔍 Search term for docs: `terraform depends_on`

1 2 3 4 2. `count` and `for_each`

🔍 Purpose:

To **create multiple resources** using a **single block**, dynamically.

grid count

- Creates **N identical** copies of a resource.
- The index is accessible via `count.index`.

```
hcl
CopyEdit
resource "aws_instance" "web" {
  count      = 3
  ami        = "ami-xyz"
  instance_type = "t2.micro"
  tags = {
    Name = "web-${count.index}"
  }
}
```

- `count = 0` skips creation.

✓ Use when: you need **N identical** instances, optionally customized by index.

for_each

- Create **one per key-value pair or item** in a map or set.
- `each.key` and `each.value` are available inside.

```
hcl
CopyEdit
resource "aws_s3_bucket" "project" {
  for_each = {
    "logs"      = "log-bucket"
    "analytics" = "analytics-bucket"
  }

  bucket = each.value
  tags = {
    Name = each.key
  }
}
```

 Use when: resources are based on **named data** (like maps, sets, or lists), and need to vary.

 Docs:

- `count`
 - `for_each`
-

3. provider

Purpose:

Explicitly select **which provider configuration** to use if multiple are defined.

```
hcl
CopyEdit
provider "aws" {
  alias  = "us_west"
```

```
region = "us-west-2"
}

resource "aws_instance" "web" {
  provider = aws.us_west
  ami      = "ami-xyz"
  instance_type = "t2.micro"
}
```

 Useful when:

- You're working across **multiple AWS regions**.
- You're managing **multi-account** infra.
- You want fine-grained control.

 Doc: Provider Meta-Argument

4. lifecycle block

This block allows fine control over how Terraform **creates**, **destroys**, or **ignores** resource changes.

create_before_destroy

hcl
CopyEdit

```
resource "aws_instance" "web" {
  lifecycle {
    create_before_destroy = true
  }
}
```

 Why:

By default, Terraform does:

Destroy → Create (to avoid duplicates)

But with this, it becomes:

Create → Test → Destroy (zero-downtime rollout)

 Use when:

- Your resource is **critical** (e.g., EC2, Load Balancers).
- You want to avoid **downtime**.
- Replacing resources that can't be modified in-place.

 Lifecycle Docs

prevent_destroy

hcl

CopyEdit

```
resource "aws_s3_bucket" "critical" {
  lifecycle {
    prevent_destroy = true
  }
}
```

 Use when:

- You want to **block accidental deletion** of critical resources (like databases, S3 buckets).
- It acts as a "**safety lock**" in production.

 Important: Once you **remove** `prevent_destroy`, Terraform **forgets** it was ever there.

replace_triggered_by

hcl

CopyEdit

```
resource "aws_instance" "web" {
  lifecycle {
    replace_triggered_by = [aws_launch_template.web_template]
  }
}
```

}

 Use when:

- You want to **trigger recreation** of a resource when another changes — even without a direct reference.
 - Great for **infra coupling**: “If template changes → recreate instance”
-

 **ignore_changes**

hcl
CopyEdit

```
resource "aws_instance" "web" {
  lifecycle {
    ignore_changes = [
      tags,
      user_data
    ]
  }
}
```

 Use when:

- Some attributes are changed **outside Terraform** (e.g., manually).
- You want to avoid **Terraform reverting them** every time.

 Overusing this can **hide drift** and is dangerous — use carefully.



Pre & Post Conditions (Brief)

- Used to **validate resource values**.
- Eg: Ensuring a tag is not empty, or instance type is valid.

hcl
CopyEdit

```

resource "aws_instance" "web" {
  precondition {
    condition      = length(var.tags) > 0
    error_message = "Tags cannot be empty."
  }
}

```

 Full Docs: Precondition and Postcondition

Real-World Use Cases Summary:

Meta-Arg	When to Use
depends_on	Manually enforce resource creation order
count	Create N copies of a resource
for_each	Create resources based on map/set/list
provider	Use multiple accounts/regions
create_before_destroy	Avoid downtime during replacements
prevent_destroy	Lock production infra from deletion
replace_triggered_by	Chain recreations across resources
ignore_changes	Prevent Terraform from reverting out-of-band changes
pre/post condition	Validate input/output before provisioning

Best Practices

- Use `prevent_destroy` for **production databases, buckets, VPCs**.
- Use `create_before_destroy` in **zero-downtime** pipelines.

- Use `ignore_changes` **sparingly**, only when absolutely necessary.
- Use `for_each` over `count` when dealing with **named resources**.
- Avoid `depends_on` unless you have **provisioners or non-obvious dependencies**.

🎯 Lecture 4: NGINX Web Server Deployment Project (Terraform)

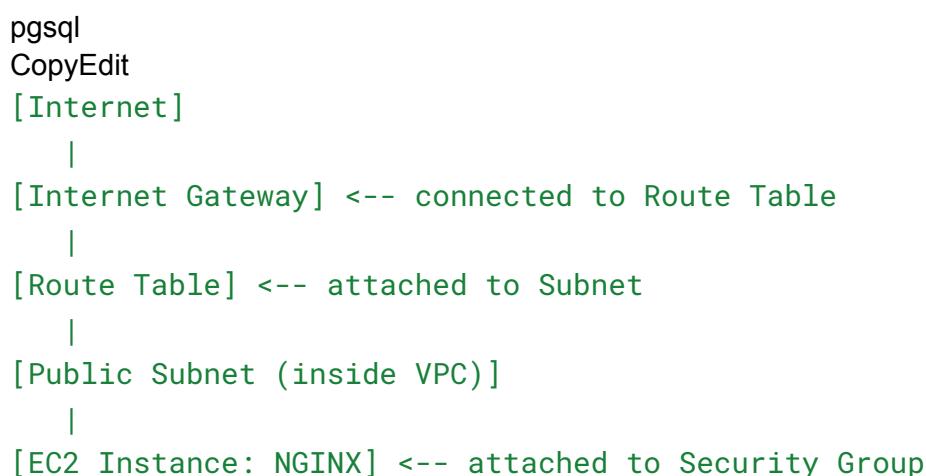
📌 🚀 Project Goal (What Are We Doing?)

We are deploying a **web server (NGINX)** on **AWS EC2**, accessible publicly via HTTP and HTTPS.

The deployment will be fully automated via Terraform, and it will involve setting up:

- A **custom VPC**
- A **public subnet** (optionally a private subnet)
- An **Internet Gateway (IGW)**
- A **Route Table** with internet access
- A **Security Group** for HTTP(80) and HTTPS(443)
- An **EC2 instance** running the **Bitnami NGINX image**
- A **test via the EC2's public IP** to ensure the web server works.

📌 📐 Architecture Diagram Overview



✓ Components and Why They're Needed

Resource	Purpose
VPC	Your own isolated network in AWS. You control IP ranges, subnets, routing.
Public Subnet	A subdivision of the VPC with access to the internet.
Internet Gateway	Allows internet access to/from your VPC.
Route Table	Contains rules to allow traffic from subnet to the internet.
Security Group	Acts like a firewall. Allows inbound HTTP (80) and HTTPS (443).
EC2 Instance	The server itself, running NGINX (via a prebuilt Bitnami image).

Why This Setup?

AWS EC2 instances don't get public internet access just by launching them. You must:

1. Put them in a **subnet** with a **route to an IGW**
2. Assign them a **public IP**
3. Attach a **security group** that allows web traffic

Terraform will automate all of this.

Optional: Private Subnet

You **can add a private subnet** to simulate a real-world scenario:

- Public subnet → Hosts NGINX (web tier)
- Private subnet → Can be used for DBs or internal services

But in this lecture, private subnet is **not required**, only public one is used.

Security Group Details

Rule Type	Port	Protocol	Source
			I

Inbound	80	TCP	0.0.0.0/0 (anywhere)
Inbound	443	TCP	0.0.0.0/0 (anywhere)
Outbound	All	All	0.0.0.0/0 (default)

EC2 Instance Details

- **AMI (Image):** Bitnami's NGINX image (free)
 - **Instance Type:** Usually `t2.micro` or `t3.micro` (free tier)
 - **User Data:** Not needed if using Bitnami image
 - **Key Pair:** Optional, for SSH access (not needed if only testing via web)
 - **Security Group:** Attach the one with ports 80/443 open
-

Testing After Deployment

Once deployed:

1. Go to AWS EC2 dashboard.
 2. Copy the **public IPv4 address** of the instance.
 3. Open it in your browser:
 - `http://<public-ip>` → Should show NGINX welcome page.
-

What to Do Next?

You can now:

- Either **try writing the Terraform code from scratch** as an exercise,
- Or continue to the next lecture where this architecture is implemented **step-by-step with Terraform**.



Key Takeaways

- This project teaches **custom VPC architecture** and how to wire it together.
- You learn what's **needed to expose a server to the internet** on AWS.
- It's a **real-world scenario** you'll encounter in production setups.
- You're using **Terraform to define and manage all infrastructure as code**.

Lecture 6



Key Concepts & AWS Cost Update



AWS IPv4 Cost Change (2024)

- As of **Feb 2024**, AWS started charging for **public IPv4 addresses** (previously free).
- **Free tier** includes:
 - 750 hours/month of public IPv4 addresses.
 - So, **1 EC2 with 1 public IP for full month = Free**.
 - Going over = small cost (around **\$0.005/hour**).
- If you're running **multiple EC2s with public IPs**, you may incur costs.



Free Tier Docs:

- AWS Free Tier: <https://aws.amazon.com/free>
-



Project Goal Recap

Deploy a public NGINX EC2 instance inside a custom VPC

This lecture covers:

- Setting up an **Internet Gateway**
 - Creating a **Route Table** and associating it with the subnet
 - Allowing public internet access via these components
-



Completed Steps (from previous lectures)

1. Created a custom VPC (**10.0.0.0/16**)

-
2. **Created a public subnet (10.0.0.0/24)**
 3. **Tagged** resources with `project`, `managed_by`, and `name`
-

New Resources in This Lecture

4 Create Internet Gateway

Terraform:

```
hcl
CopyEdit
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name      = "06-resources-main"
    Project   = "06-resources"
    ManagedBy = "Terraform"
  }
}
```

Explanation:

- Internet Gateways allow resources in the VPC to access the **public internet**.
- Must be **attached to the VPC**.

Docs:

- `aws_internet_gateway`
-

5 Create Public Route Table

Terraform:

```
hcl
```

CopyEdit

```
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0" # Route all traffic
    gateway_id = aws_internet_gateway.main.id
  }

  tags = {
    Name      = "06-resources-public"
    Project   = "06-resources"
    ManagedBy = "Terraform"
  }
}
```

 **Explanation:**

- A Route Table defines **where traffic should go**.
- **0.0.0.0/0** is a **default route to the internet**, using the IGW.

 **Docs:**

- [aws_route_table](#)
-

6 Associate Route Table with Subnet

Terraform:

hcl

CopyEdit

```
resource "aws_route_table_association" "public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

 **Explanation:**

- This **binds the route table to the subnet**, making it a **public subnet** (i.e., has internet access).
- No tags needed for association—it's a **link**, not a standalone resource.

 Docs:

- [aws_route_table_association](#)
-

Tags Discussion

Tags used:

```
hcl
CopyEdit
tags = {
  Name      = "06-resources-<resource>"
  Project   = "06-resources"
  ManagedBy = "Terraform"
}
```

Why tags matter:

- **Visibility** in AWS Console
- Easy **filtering/searching**
- **Cost tracking** (e.g., by project/team)

Common tags:

- `environment, cost_center, team`
-

Terraform Commands Run

```
bash
CopyEdit
terraform fmt          # Format the files
```

```
terraform plan          # Preview changes  
terraform apply         # Apply changes  
terraform apply -auto-approve # Apply without confirmation
```



Tip:

- Use `terraform plan` to **confirm** what will change.
 - `terraform apply -auto-approve` skips the `yes` prompt.
-



Verifying in AWS Console

1. Go to **VPC Dashboard**
2. View:
 - **VPCs** → your `06-resources` VPC
 - **Subnets** → your `public` subnet
 - **Route Tables** → should show:
 - Route: `0.0.0.0/0` → `Internet Gateway`
 - Association to your public subnet
3. Go to **Internet Gateways** → should be attached to VPC



AWS Console shows a **graphical resource map**, useful for visual inspection.



Key Learnings & Pro Tips

Concept	Explanation
Public Subnet	A subnet becomes public when associated with a route table that has a route to an Internet Gateway .
IGW	Allows EC2 to connect to the internet. Must be attached to a VPC .

Route Table Controls where traffic goes (e.g., default to $0.0.0.0/0 \rightarrow$ IGW).

Association Links a route table to a specific subnet.

Tags Metadata for filtering, billing, and UI clarity. Always tag!

What's Next?

You now have:

- A **VPC**
- A **Public Subnet**
- **Internet Access** set up via **Route Table + IGW**

 Next lecture will likely cover:

- **Creating a Security Group**
- **Launching EC2 instance**
- **Attaching Public IP**
- **Testing NGINX via browser**

Lecture 7: Refactoring Tags with `locals` and `merge` Function in Terraform

Goal of the Lecture

- Avoid **code duplication** for resource tagging.
 - Introduce `locals` for **maintainability**.
 - Use Terraform's `merge` function to combine **common tags + custom tags**.
 - Compare local tagging vs provider-level tagging.
-

Concept Introduced: `locals` Block

➤ What are `locals` in Terraform?

- Used to **define local variables** inside a configuration.
- Helpful for:
 - **Avoiding duplication.**
 - Improving **code readability** and **reusability**.
 - Acting as **intermediate values**, not external inputs.

Syntax:

```
hcl
CopyEdit
locals {
  common_tags = {
    Project      = "TerraformBootcamp"
    ManagedBy   = "Terraform"
  }
}
```

- `locals` (plural) is the **block declaration**.

- `local` (singular) is how you **reference** it later:

```
hcl
CopyEdit
tags = local.common_tags
```

Before Refactor:

```
hcl
CopyEdit
tags = {
  Project    = "TerraformBootcamp"
  ManagedBy  = "Terraform"
}
```

 Repeated in **multiple resource blocks** → BAD PRACTICE 

After Refactor Using `locals`:

Apply `local.common_tags`:

```
hcl
CopyEdit
tags = local.common_tags
```

Adding Resource-Specific Tags via `merge()`

➤ `merge()` Function

- Combines multiple maps (dictionaries).
- If duplicate keys exist, the **last one wins**.

Syntax:

```
hcl
CopyEdit
tags = merge(
  local.common_tags,
```

```
{  
  Name = "06-resources-main"  
}  
)
```

🎯 Now, all resources can have:

- **Common tags** from `locals`.
 - **Resource-specific tag** like `Name`.
-

✓ Example Full Tag Block for a Resource:

```
hcl  
CopyEdit  
tags = merge(  
  local.common_tags,  
  {  
    Name = "06-resources-main"  
  }  
)
```

⊕ 🎒 Adding More Tags (e.g., CostCenter)

To add shared tags to all resources:

```
hcl  
CopyEdit  
locals {  
  common_tags = {  
    Project      = "TerraformBootcamp"  
    ManagedBy    = "Terraform"  
    CostCenter   = "1234"  
  }  
}
```

After `terraform plan`, you'll see:

bash

CopyEdit

```
# aws_vpc.main will be updated in-place
~ tags = {
    + CostCenter = "1234"
}
```

Which Resources Had Tags Updated?

-  VPC
-  Subnet
-  Internet Gateway
-  Route Table

 *Route Table Association* : Cannot be tagged (not supported).

Tip: Terraform `plan` Helps Catch Mistakes

- Example mistake: overwriting tag `Name` in multiple resources with same value ([06-resources](#)).
 - Fixed by using `merge(local.common_tags, { Name = "resource-specific-name" })`.
-

`locals` vs `provider.default_tags`

Option 1: `locals` (Used in lecture)

Pros:

- More flexible with **per-resource merging**.
- Easy to refactor & customize.

Cons:

- Must add `tags = merge(...)` block to **every resource** manually.

Option 2: provider block with `default_tags`

hcl
CopyEdit

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

  default_tags {
    tags = {
      Project    = "TerraformBootcamp"
      ManagedBy = "Terraform"
    }
  }
}
```

Pros:

- Automatically applies to **all AWS resources** in the config.
- No need to repeat tag blocks.

Cons:

- **No support for resource-specific tags** unless explicitly merged again.
- Applies to **all resources**, which may be undesired in some cases.

Code Summary (Final Tags Refactor)

hcl
CopyEdit

```
locals {
  common_tags = {
    Project      = "TerraformBootcamp"
    ManagedBy   = "Terraform"
    CostCenter  = "1234"
  }
}
```

```
resource "aws_vpc" "main" {
    ...
    tags = merge(
        local.common_tags,
        { Name = "06-resources-main" }
    )
}
```

✓ Terraform Commands Used

- `terraform fmt` → format code.
 - `terraform plan` → preview changes.
 - `terraform apply` → apply the infrastructure.
 - `terraform apply -auto-approve` → skip confirmation.
 - `terraform plan` after refactor → ✓ no infrastructure drift (infra matches config).
-

✓ Updated Progress in Project

✓ Step 3 completed in the README:

- Deployed internet gateway
 - Set up a route table with a route to the internet gateway
 - Associated the route table with the public subnet
-

⌚ Final Notes

- `locals + merge()` is a **powerful pattern** for DRY (Don't Repeat Yourself) coding in Terraform.
- Helps maintain large-scale infrastructure cleanly.

- We'll revisit `locals` and Terraform functions later in depth.
- `default_tags` in `provider` is an alternative, useful for **universal tagging**.

Lecture 8 – Deploying EC2 Instance with Terraform (God-Level Notes)

Context Recap (So Far)

We have already:

1. **Created a VPC**
2. **Created a Public Subnet**
3. **Deployed an Internet Gateway**
4. **Created a Route Table & Association**
5. **Refactored Tags using `locals + merge()` function** to reduce repetition

All these were **networking pre-requisites** to host an EC2 instance.

Goal of This Lecture:

Deploy an EC2 instance inside the public subnet, assign a public IP, and prep it for hosting a web server (Nginx from Bitnami image).

Step-by-Step Breakdown:

1. Create a new file for compute-related resources:

```
hcl
CopyEdit
// compute.tf
resource "aws_instance" "web" {
    ami           = "ami-xyz123"      # Replace with correct AMD64
image ID
    instance_type = "t2.micro"        # Free-tier eligible
    subnet_id     = aws_subnet.public.id
    associate_public_ip_address = true
```

```

root_block_device {
  volume_size          = 10
  volume_type          = "gp3"
  delete_on_termination = true
}

tags = merge(
  local.common_tags,
  {
    Name = "06-resources-web"
  }
)
}

```

2. Find a Valid Public AMI

- Use Ubuntu EC2 Image Locator
 - Choose architecture: **AMD64** (not ARM)
 - Choose your **region** (e.g., `eu-west-1`)
 - Copy the correct AMI ID and replace in your Terraform code
-

3. Understand EC2 Parameters

Parameter	Meaning
<code>ami</code>	The OS image used to launch instance (Ubuntu, Amazon Linux, etc.)
<code>instance_type</code>	The compute configuration (e.g., <code>t2.micro</code> for free tier)
<code>subnet_id</code>	The subnet the instance will reside in
<code>associate_public_ip_address</code>	Ensures it's accessible from the internet
<code>root_block_device</code>	Config for the root volume (disk)



Search terms for documentation:

- "aws_instance":
<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>
 - "aws_subnet":
<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/subnet>
 - "terraform root_block_device": Check EC2 instance docs above
-

4. Root Block Device Settings

hcl

CopyEdit

```
root_block_device {  
    volume_size          = 10           # Size in GB  
    volume_type          = "gp3"        # General purpose SSD (more  
    modern)  
    delete_on_termination = true       # Auto-delete on instance  
    removal  
}
```

 Why gp3 over gp2?

- Better performance and same cost
 - Still free-tier eligible
-

5. Plan, Apply, Test

bash

CopyEdit

```
terraform fmt          # Format code  
terraform plan         # Preview changes  
terraform apply        # Deploy EC2 instance
```

 If you get an error like:

InvalidParameterValue: The architecture 'arm64' does not
match the instance type 't2.micro'

 You probably picked an **ARM64 AMI** — switch to **AMD64** image.

6. Verify in AWS Console

- Go to **EC2 > Instances**
 - See the new instance running
 - Check for:
 - **Public IPv4 address**
 - **Security group (default)**
 - **AMI used**
 - You **won't** be able to access via browser yet (no inbound rule on port 80)
-

7. Remove EC2 Instance Safely (Temporary)

If you're pausing here:

```
hcl
CopyEdit
# Comment out aws_instance "web" block

bash
CopyEdit
terraform plan      # Will show 1 to destroy
terraform apply      # Confirm and destroy only the instance
```

Why?

- Prevent accidental **750-hour EC2 usage**
 - Billing protection if you're not continuing immediately
-

8. Destroy vs. Comment + Apply

Option	Effect
--------	--------

`terraform` **Destroy** Destroys all resources in state file

Comment + **apply** Destroys only the resources you've removed



Key Learnings from This Lecture:

Concept	Importance
Public AMI selection	Must match your architecture + region
Instance sizing	T2.micro or T3.micro for free tier, region dependent
<code>root_block_device config</code>	Ensures proper cleanup, avoids storage cost
Security best practice	No dangling EC2s or disks
<code>merge() + locals</code>	DRY coding for tags, cleaner & maintainable infrastructure



Useful Terraform Docs

- `aws_instance`
 - ubuntu EC2 AMIs
 - default tags via provider
-



Summary

This lecture was about deploying a minimal, publicly accessible EC2 instance through Terraform, building on the networking stack we already created. We also ensured that we:

- Clean up idle resources
- Avoid hidden costs
- Work toward production-grade automation via reusable tagging and modular infrastructure

LECTURE 9: Understanding Default Security Groups in AWS EC2

Objective of the Lecture

This lecture dives deeper into:

- What happens when you create an EC2 instance **without specifying a Security Group**.
 - Understanding the **default security group** automatically assigned by the VPC.
 - **How internal EC2 communication works via security groups**.
 - Laying the groundwork for **creating custom security groups** in the next lecture.
-



High-Level Flow

1. **Uncomment EC2 resource block in Terraform** to recreate the EC2 instance.
 2. **Run `terraform apply`** to provision the instance.
 3. **Inspect the EC2 instance in AWS Console** — focus on the "Security" tab.
 4. Learn how AWS auto-assigns a **default security group** if none is provided.
 5. Deep dive into how this **default SG enables intra-VPC EC2 communication**.
 6. Look at how **rules based on security group references** work.
 7. Discuss best practices for **scoping security rules by SG instead of CIDR/IP**.
-



What We Are Doing

Step 1: Recreating EC2 Instance (via Terraform)

You uncomment the EC2 resource in your `.tf` file:

hcl
CopyEdit

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

Then:

```
bash
CopyEdit
terraform apply
```

AWS provisions the EC2 instance. No security group is explicitly defined.

✓ Step 2: What Happens Internally?

Despite not providing a `vpc_security_group_ids` block in the resource, the instance still gets a **security group**. How?

By default, AWS attaches the VPC's default security group to any EC2 instance that doesn't have one explicitly assigned.

🔍 Understanding the Default Security Group

🔒 What Is It?

- Automatically created when a VPC is created.
- Attached **by default** to EC2 instances launched in that VPC, **unless overridden**.

🔍 Where to View It:

Go to EC2 instance in AWS Console → Security tab → Click the attached security group → You'll see something like:

```
vbnet
CopyEdit
Group Name: default
Group ID: sg-xxxx
VPC: vpc-xxxx
```



Inbound Rules of Default SG

Default SG usually has this:

```
pgsql
CopyEdit
Type: All traffic
Protocol: All
Port Range: All
Source: sg-xxxx (itself)
```

This means:

- **All inbound traffic is only allowed if it comes from other resources that are also using this security group.**
 - Not from the public internet.
 - Not from your laptop.
 - Just from other EC2s using the same SG.
-



What This Enables

Intra-VPC EC2 Communication!

If you have 2 EC2s:

- **Instance A** uses default SG.
 - **Instance B** uses default SG.
 - Then: they can talk to each other over **any port** (because of the rule allowing traffic from the same SG).
-



Security Group Rules – Key Concepts

Concept	Description
SG ID	Identifier of the security group. Example: <code>sg-xxxxxxxx</code> .
SG Rule ID	Unique ID for each inbound/outbound rule.
Referencing SG in Source	You can allow traffic from another SG , not just from IPs.
Default Rule	Allows traffic only from same SG — great for internal communication.



Use Case: Scoped Database Access

Let's say:

- You launch a **MySQL RDS database**.
- You want **only EC2s from a specific group to talk to it**.

Then you can:

- Create an SG for the DB that **allows inbound MySQL (port 3306)**.
- Set the **source as the EC2 SG ID**, not a public IP or CIDR block.

This approach:

- Is **dynamic** (IP changes don't break it).
 - Is **secure** (only AWS-internal traffic from whitelisted EC2s allowed).
 - Follows the **principle of least privilege**.
-



Cleanup Recommendation

Before moving to the next lecture:

- Either **comment out the EC2 resource**.
- Or **destroy** it via:

bash
CopyEdit
`terraform destroy -target aws_instance.web`

Avoid burning your free tier EC2 hours if you're pausing the course!

✓ Key Takeaways

Key Concept	Meaning
● Security Groups	Act as virtual firewalls controlling traffic to/from EC2.
● Default SG	Auto-attached to instances if none specified. Allows internal traffic between instances sharing the same SG.
● Referencing SGs	A powerful way to scope access between resources without exposing them publicly .
● No SG = No Access	An EC2 with no SG at all (not even default) is unreachable from anything.
● Best Practice	Use SG-to-SG referencing , especially in microservice or multi-tier architecture.
✓ Resource Hygiene	Clean up test EC2s to save free tier hours!

➡ SOON Coming Up Next

In the **next lecture**, you'll learn how to:

- Define **your own security group**.
- Attach it to your EC2 instance.
- Control **specific ports (like 22 or 80)** and access sources.

LECTURE 10: Creating Public Security Group & Associating It with EC2

Goal

- Create a **custom Security Group** to allow public (internet) access to EC2.
 - Open ports **80 (HTTP)** and **443 (HTTPS)**.
 - Attach this Security Group to your existing EC2 instance.
 - Understand how **Terraform handles in-place updates**.
 - Observe **dependency resolution and resource creation order** in Terraform.
-

Why We're Doing This

- By default, EC2 instances with the default Security Group **are not accessible from the internet**.
 - Default SG only allows **internal traffic between instances sharing the same SG**.
 - If we want to **access the EC2 via a browser**, we need to explicitly allow public inbound traffic on specific ports (80, 443).
-

Terraform Steps – Full Flow

Step 1: Create Security Group

hcl

CopyEdit

```
resource "aws_security_group" "public_http_traffic" {
    name          = "public_http_traffic"
    description   = "Allow HTTP and HTTPS traffic"
    vpc_id        = aws_vpc.main.id

    tags = {
        Name = "06-resources-sg"
```

```
    }
}
```

- **Resource Type:** `aws_security_group`
 - **VPC-bound:** All SGs must belong to a VPC.
 - **Tags** help you identify the SG in AWS Console.
-

Step 2: Create Ingress Rules (open ports)

hcl

CopyEdit

```
resource "aws_vpc_security_group_ingress_rule" "http" {
  security_group_id = aws_security_group.public_http_traffic.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port          = 80
  to_port            = 80
  ip_protocol        = "tcp"
}

resource "aws_vpc_security_group_ingress_rule" "https" {
  security_group_id = aws_security_group.public_http_traffic.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port          = 443
  to_port            = 443
  ip_protocol        = "tcp"
}
```

- **Resource Type:** `aws_vpc_security_group_ingress_rule`
 - **0.0.0.0/0:** Means **from anywhere**.
 - Ports:
 - 80 = HTTP (insecure)
 - 443 = HTTPS (secure)
-

Why Not SSH (Port 22)?

- **SSH (port 22)** is typically **not added intentionally** to avoid security risks.
 - There are better, more secure ways (e.g., AWS SSM) to access EC2 instances if needed.
-

Step 3: Associate Security Group with EC2

In your EC2 resource block:

```
h
CopyEdit
resource "aws_instance" "web" {
    ami                      = "ami-xxxxxxxx"
    instance_type            = "t2.micro"
    subnet_id                = aws_subnet.public.id
    associate_public_ip_address = true

    vpc_security_group_ids = [
        aws_security_group.public_http_traffic.id
    ]

    tags = {
        Name = "06-resources-ec2"
    }
}
```

- **vpc_security_group_ids**: List of **SG IDs** to associate.
 - Updating this field **does NOT recreate the EC2 instance** – it updates it **in place**.
-

Terraform Behavior: Apply Flow

When you run:

```
bash
CopyEdit
terraform apply
```

Terraform:

1. **Creates the new Security Group.**
2. Creates the **two ingress rules** for HTTP and HTTPS.
3. **Updates the EC2 instance in place**, replacing the default SG with your new custom SG.

 No instance termination. Just an in-place network config change.

Visual: Creation Order

Terraform shows:

-  to add
-  to modify
-  to destroy

Output will show:

text

CopyEdit

Plan: 3 to add, 1 to change, 0 to destroy.

Validation: Confirm in AWS Console

Go to:

- EC2 → Instances → Your instance → **Security tab**

You should now see:

- Security Group: `public_http_traffic`
- Inbound rules:

- HTTP: Port 80 from 0.0.0.0/0
- HTTPS: Port 443 from 0.0.0.0/0

✓ That means you can now access your EC2 instance via its public IP in the browser!

✓ Summary Table

Concept	Description
<code>aws_security_group</code>	Defines a SG in AWS. Bound to a VPC.
<code>aws_vpc_security_group_ingress_rule</code>	Adds an inbound rule to allow traffic from public internet on specified port.
<code>vpc_security_group_ids</code>	Tells the EC2 which SG(s) to attach to.
In-place update	Terraform changes SG association without destroying EC2.
No SSH	Port 22 is not opened for better security.
Tags	Help identify SG and EC2 easily in Console.

⌚ Optional Cleanup

If you're **pausing development**, it's good practice to **tear down the EC2** to save free-tier hours:

```
bash
CopyEdit
# Comment out the EC2 block
terraform apply
```

Or destroy just the instance:

```
bash
CopyEdit
terraform destroy -target aws_instance.web
```



Resources

- Terraform Security Group Docs
 - Ingress Rule Docs
 - VPC and SG Concepts – AWS Docs
-



Next Steps

In the next lecture, you'll likely:

- Add **user data** (startup scripts) to install and serve NGINX automatically.
- Maybe attach an **Elastic IP** or work with **outputs**.

Lecture 11 – Launching EC2 with Public Bitnami NGINX AMI (Manual Method)

Objective:

Replace your previous EC2 instance (default Amazon Linux or Ubuntu) with a **publicly available Bitnami NGINX AMI** from the **AWS Marketplace** — specifically one that is **free-tier eligible**.

Step-by-Step Breakdown

1. Go to AWS Console

- Log in to the AWS Console
 - Navigate to: **EC2 > Instances** (make sure you're in the correct region)
-

2. Find the Correct Public NGINX AMI

Navigate:

- On the left-hand sidebar:
 - Click “**AMIs**” under “**Images**”
 - Then click “**Public images**” or go to “**AMI Catalog**” (depending on the UI version)

Filter:

- Click “**AWS Marketplace AMIs**”
- In the search box, **type: nginx**
- You will likely see **500+ results**

Narrowing Down:

- On the **left side**, under “**Pricing Model**”, click:

- **Free**
 - This filters only **free-tier eligible** AMIs

Select:

- Look for:
 - **Bitnami NGINX Open Source** image
 - Provided by **Bitnami**
 - **Verified**
 - Typically titled:

"Nginx Open Source Certified by Bitnami and Automatic"

 There may be multiple versions. Always check for:

- **Free Tier Eligible**
- **Verified provider**
- **Updated recently**

3. Copy the AMI ID

- Once you find the correct AMI, **note/copy the AMI ID**
 - Format: `ami-xxxxxxxxxxxxxxxxxx`
 - This ID is **region-specific**, so it will differ per region.

Example:

`ami-0abcdef1234567890` (just a placeholder)

4. Launch EC2 Instance Using This AMI

Go to **EC2 > Launch Instance**, and follow these steps:

Step 1: Name and AMI

- **Name:** e.g. `nginx-bitnami-instance`
- **AMI:**
 - Choose “**My AMIs**” > “Owned by me” > “**AMIs from AWS Marketplace**”
 - Paste the **AMI ID** you copied
 - It should show you the Bitnami NGINX details
 - Click **Select**

Step 2: Instance Type

- Choose: `t2.micro` (Free tier eligible)

Step 3: Key Pair

- Use existing key pair or create a new one.
- This is required for **SSH access** if needed.

Step 4: Network Settings

- **VPC:** Select your previously created VPC
- **Subnet:** Select the **public subnet**
- **Auto-assign Public IP:** Enable (to access via internet)

Step 5: Firewall (Security Group)

- Use your **existing security group** or **create a new one**:
 - **Allow HTTP (port 80)**
 - **Allow HTTPS (port 443)**
 - **Allow SSH (port 22)** (for admin access)

Make sure these rules allow access from `0.0.0.0/0` (worldwide), unless you're restricting.

Step 6: Storage (Optional)

- The Bitnami image may predefine required storage (usually 10GB+)
- You can accept default, or increase if needed

Final Step: Launch

- Click **Launch Instance**
 - Wait until **Instance State** becomes **Running**
-

5. Test NGINX Deployment

- Go to **EC2 > Instances**
- Copy the **Public IPv4 address**

Paste in your browser:

```
cpp
CopyEdit
http://<Public-IP>
```

-
- You should see a **Bitnami NGINX Welcome Page**

 This confirms the EC2 instance is running and NGINX is properly installed.

Important Concepts from This Lecture

AMI (Amazon Machine Image)

- **Public AMIs** are pre-built server environments.

- AWS Marketplace AMIs offer vendor-certified images (e.g. Bitnami, Ubuntu, Jenkins, etc.)
- Not all AMIs are **free** — you must check pricing **before launch**

Bitnami AMI Features

- Bitnami packages often come with:
 - Preconfigured software
 - Best practices baked in (security, patches)
 - Admin tools & custom splash pages

Dependency Reminder

- You must **launch** EC2 in a **public subnet**
 - Subnet must be attached to a **Route Table with an Internet Gateway**
 - Security Group must allow **HTTP traffic**
-

Common Mistakes to Avoid

Mistake	Consequence
Using private subnet	No internet access, can't reach from browser
Not assigning Public IP	EC2 won't be reachable
Security Group missing port 80	HTTP won't work
Choosing paid AMI	You might incur unexpected charges

Suggested Practice

For better understanding, try launching:

1. Another EC2 instance using a **non-nginx** image (like Apache)

2. One in a **private subnet** (to see why public access fails)
3. Use **User Data** to install nginx manually instead of using Bitnami

Lecture 12 — Full Lifecycle: Managing and Destroying Infrastructure with Terraform

Objective:

- Inspect what Terraform is managing.
 - Learn how it handles *external changes* (drift).
 - Introduce `ignore_changes` to handle team/process edits safely.
 - Destroy all provisioned resources cleanly using `terraform destroy`.
-

1. Inspect Terraform-Managed Resources

bash

Copy>Edit

```
terraform state list
```

What this does:

- Lists *all AWS resources* currently managed by the Terraform project.
- Gives confidence that the full stack is under Terraform's control.

Example Output:

- `aws_instance.web`
- `aws_internet_gateway.main`
- `aws_route_table.main`
- `aws_route_table_association.main`
- `aws_security_group.public_http_traffic`
- `aws_security_group_rule.http`

- `aws_security_group_rule.https`
- `aws_subnet.public`
- `aws_vpc.main`

 **Insight:** Even a simple EC2 setup ends up requiring 8–10 resources.

2. Terraform Drift Detection & Refresh Behavior

When you run:

```
bash
CopyEdit
terraform apply
```

Terraform automatically performs:

-  **State Refresh** — fetches the *current actual state* of resources in AWS.
 -  **Compares actual state to declared config**
 -  **Plans changes** if it finds differences.
-

3. Example of Drift (Manual Change Outside Terraform)

Scenario: You manually add a tag to the EC2 instance via AWS Console:

```
json
CopyEdit
Key: team
Value: finance
```

Then run:

```
bash
CopyEdit
terraform apply
```

Terraform will detect that:

- This tag **does not exist** in the Terraform configuration.
 - Result: Terraform **plans to remove** it from the resource.
-

4. Avoiding Terraform Overwrites (Handling External Changes)

Use the `lifecycle` block with `ignore_changes`:

```
hcl
CopyEdit
resource "aws_instance" "web" {
    ...
    lifecycle {
        ignore_changes = [tags]
    }
}
```

Purpose:

- Tells Terraform to **ignore drift** related to the `tags` attribute.
- Useful when:
 - Other teams/tools update the resource.
 - Tags are injected automatically by AWS organizations or pipelines.

Best Practice:

- Avoid `ignore_changes` unless absolutely necessary.
 - Prefer managing **everything via Terraform** when possible.
-

5. Revert Manual Changes (Optional Cleanup)

- Remove `ignore_changes`
- Run:

```
bash
CopyEdit
terraform apply --auto-approve
```

⌚ Result: Terraform removes externally added tags and **realigns the resource** with the `.tf` config.

🔥 6. Destroy All Resources with Terraform

Option 1:

```
bash
CopyEdit
terraform destroy
```

Option 2 (alias):

```
bash
CopyEdit
terraform apply -destroy
```

- Terraform again:
 - **Refreshes state**
 - **Calculates plan**
 - **Destroys all resources** under management
- You'll be prompted to confirm with `yes`.

Example:

`Plan: 0 to add, 0 to change, 9 to destroy`

📅 Timeline Example:

- EC2 instance: Took ~51 seconds to terminate
 - Remaining infrastructure destroyed near-instantly
-

7. Validation of Destruction

 After destroy completes:

- Run:

```
bash
CopyEdit
terraform state list
```

Output: **empty**

- Check AWS Console:
 - EC2 instance: **terminated**
 - VPC: **deleted**
 - Subnet, IGW, route tables, SGs: **gone**
-

Key Concepts Recap

Concept	Summary
<code>terraform state list</code>	Shows what's currently managed
State Refresh	Auto-fetched before apply/destroy
Drift	Changes outside of Terraform
<code>ignore_changes</code>	Avoid overwriting external edits
<code>terraform destroy</code>	Cleans up everything created by this config

Best Practice

All changes should go through Terraform when possible



Bonus Tips

- 💡 **Check for leftovers** manually in AWS Console (e.g. orphaned volumes, elastic IPs).
 - 🛡️ Use `terraform plan` before `apply` or `destroy` in production workflows.
 - 📄 Use `terraform fmt` to keep config clean and readable.
-



README Update

Mark Step 6 (Launch EC2 with NGINX AMI) and Step 7 (Destroy Infrastructure) as **Done**.



Summary

You've now:

- Created AWS infrastructure from scratch
- Modified and detected drift
- Ignored safe manual changes
- Cleanly destroyed every provisioned resource

This wraps up your **first full Terraform lifecycle project**.

Next up, we'll dive deeper into **modules**, **remote state**, and **multi-environment setups**.

s3 static website

Lecture 1



PROJECT GOAL:

Deploy a static website on **Amazon S3**, fully managed via **Terraform**.



PROJECT OUTCOMES:

By the end of the project:

- An S3 bucket will host HTML files (static website).
 - Initially, files are uploaded manually via AWS Console.
 - Later, Terraform (`aws_s3_object`) will upload them.
 - Public Access block will be disabled.
 - A **bucket policy** will be created to allow public read access (`s3:GetObject`).
 - Terraform will configure the **S3 static website hosting** settings.
 - The **website endpoint** will be output from Terraform.
-



PROJECT SETUP

📁 Folder Structure

Create a new project folder:

```
bash
CopyEdit
mkdir proj01-s3-static-website
cd proj01-s3-static-website
```

-
-

provider.tf

Terraform Configuration

hcl
CopyEdit

```
terraform {
  required_version = ">= 1.7.0, < 2.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.0.0, < 6.0.0"
    }
    random = {
      source  = "hashicorp/random"
      version = ">= 3.0.0, < 4.0.0"
    }
  }
}
```

AWS Provider Block

hcl
CopyEdit

```
provider "aws" {
  region = "eu-west-1" # Or your preferred region
}
```

s3.tf

STEP 1: Random Suffix for Unique Bucket Name

hcl
CopyEdit

```
resource "random_id" "bucket_suffix" {
  byte_length = 4
}
```

- Uses 4-byte (32-bit) random hex string.

- Prevents `BucketAlreadyExists` error due to S3 global naming constraint.
 - Ensures uniqueness across accounts/projects.
-

STEP 2: Create S3 Bucket for Static Website

hcl
CopyEdit

```
resource "aws_s3_bucket" "static_website" {  
  bucket =  
    "terraform-course-project-1-${random_id.bucket_suffix.hex}"  
}
```

This will generate bucket names like:

CopyEdit

```
terraform-course-project-1-a1b2c3d4
```

- S3 bucket names must be **globally unique**, lowercase, no spaces, no underscores.
-

VERIFICATION STEPS

Format the Code

bash
CopyEdit

```
terraform fmt
```

 Does not need `terraform init` first. Just formats files.

Initialize the Project

bash
CopyEdit

```
terraform init
```

 Downloads providers: `aws`, `random`

Plan

bash
CopyEdit
`terraform plan`

Expected:

- 1 x `aws_s3_bucket`
 - 1 x `random_id`
-

Apply

bash
CopyEdit
`terraform apply`

Confirm with `yes`

Post-Deployment Check

1. Go to AWS Console → S3.

You should see a bucket:

CopyEdit
`terraform-course-project-1-xxxxxxxx`

- 2.
3. Bucket will be empty initially (no objects uploaded yet).
4. Later steps will:
 - Upload files
 - Set bucket policy

- Configure website hosting
-



Summary of Resources Created

Resource	Purpose
<code>random_id.bucket_suffi x</code>	Generates a random hex string to ensure bucket name uniqueness
<code>aws_s3_bucket.static_w ebsite</code>	Creates the actual S3 bucket with a unique name for hosting static website files



What's Coming Next

- Upload `index.html` and `error.html` manually first.
- Later: Automate file upload via `aws_s3_object`.
- Add website configuration block.
- Create public-read bucket policy.
- Output static site endpoint from Terraform.



Lecture 2 – Configuring Public Access and Bucket Policy for S3 Static Website



Objective

We already created an S3 bucket in Lecture 1. In this lecture, we:

- Disable the **Public Access Block** (by default it blocks all public access).
 - Add a **Bucket Policy** that allows public users to **read objects** (`GetObject`) from the bucket.
 - Learn how to use `jsonencode` for writing IAM policies in Terraform.
 - Discuss **Terraform documentation** navigation and debugging unexpected errors like `AccessDenied`.
-



File: `s3.tf` (continued)



Step 1: Disable Public Access Block



Why?

By default, S3 blocks all public access to buckets. To make the website accessible, we must **explicitly disable all public access blocks**.



Full Resource

hcl

CopyEdit

```
resource "aws_s3_bucket_public_access_block" "static_website" {
  bucket = aws_s3_bucket.static_website.id

  block_public_acls      = false
  block_public_policy     = false
```

```
    ignore_public_acls      = false
    restrict_public_buckets = false
}
```

All 4 values must be set to `false` to allow public access.

Docs Reference:

- AWS S3 Bucket Public Access Block – Terraform Registry
-

Step 2: Create Bucket Policy for Public Read Access

Why?

Even with public access allowed, you need to define a **Bucket Policy** that explicitly allows unauthenticated users (`Principal = "*"`) to **read objects**.

Full Resource with `jsonencode`

hcl

CopyEdit

```
resource "aws_s3_bucket_policy" "static_website_public_read" {
  bucket = aws_s3_bucket.static_website.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid      = "PublicReadGetObject"
        Effect   = "Allow"
        Principal = "*"
        Action   = "s3:GetObject"
        Resource  = "${aws_s3_bucket.static_website.arn}/*"
      }
    ]
  })
}
```

Key Details:

- `jsonencode({ ... })` converts HCL syntax to a valid JSON policy.
 - `Principal = "*"` means the policy applies to **everyone** (unauthenticated users).
 - `Resource = "${bucket.arn}/*"` gives access to **all objects** in the bucket (not the bucket itself).
 - `Action = "s3:GetObject"` allows object read only (no upload, list, delete, etc).
-

Terraform Commands

Format

```
bash
CopyEdit
terraform fmt
```

Plan

```
bash
CopyEdit
terraform plan
```

Confirms that the `arn` is interpolated correctly.

Apply

```
bash
CopyEdit
terraform apply
```

 You might see this error:

```
pgsql
CopyEdit
Error putting S3 policy: AccessDenied: Access Denied
```

Troubleshooting: Access Denied on First Apply

Solution: Re-run Apply

```
bash
CopyEdit
terraform apply
```

- Often caused by **race conditions**.
 - Terraform may try to attach the policy **before** the access block has been disabled.
 - A second `apply` usually works because the access block resource is already in place.
-

Bonus: Dependency Management (Optional)

You could force a dependency to avoid race conditions:

```
h
CopyEdit
resource "aws_s3_bucket_policy" "static_website_public_read" {
  bucket = aws_s3_bucket.static_website.id
  depends_on = [
    aws_s3_bucket_public_access_block.static_website
  ]

  policy = jsonencode({ ... })
}
```

This ensures policy is applied **after** public access block has been disabled.

Permissions Debugging

If you see persistent `AccessDenied` errors:

1. Go to AWS Console → IAM → Users → Your User
2. Click on **Permissions**

3. If using `AdministratorAccess`, it should already include:

- `s3:PutBucketPolicy`
- `s3:GetBucketPolicy`
- Full `s3:*` permissions

 You can check:

- Click on `AdministratorAccess` → `S3` → Scroll to see individual permissions
 - Confirm `PutBucketPolicy` is allowed
-



Terraform Registry Best Practices

To explore resources:

1. Visit <https://registry.terraform.io>
 2. Search for:
 - `aws_s3_bucket_public_access_block`
 - `aws_s3_bucket_policy`
 3. Always choose the **official provider** from `hashicorp/aws`
-



Final State After This Lecture

Resource	Description
<code>aws_s3_bucket_public_access_block.static_website</code>	Disables public access block so public policies are allowed
<code>aws_s3_bucket_policy.static_website_public_read</code>	Enables <code>s3:GetObject</code> for all users to access website files

Summary

What We Did

Disabled S3 Public Access Block

Created a Bucket Policy with
`s3:GetObject`

Used `jsonencode()` for inline policy writing

Handled possible `AccessDenied` errors

Learned how to use the Terraform registry effectively

Why

So public access policies will work

To allow public read access to website files

Clean and Terraform-native way to generate JSON

Caused by race conditions between resource creation

For reference and correct usage patterns

Important Tips

-  Always disable all 4 public access settings.
-  Use `#{bucket.arn}/*` to refer to all files inside the bucket.
-  Don't use "`s3:*`" unless you want full access (security risk).
-  Terraform may show transient permission errors – reapply to resolve.
-  `depends_on` can help enforce execution order.

Lecture 3: Static Website Hosting with S3 – Part 3



Goal: Finalize static website configuration and test the setup.



Overview

In this lecture, we:

- Configure the **S3 static website hosting block** using Terraform.
 - Create and manually upload `index.html` and `error.html`.
 - Test the hosted site via the generated website URL.
 - Understand the shortcomings of manual deployment and set the stage for automating this.
-



Terraform Code Breakdown

1. Configure Static Website Hosting

We use `aws_s3_bucket_website_configuration` to tell S3 how to behave as a static site.

♦ Resource Block

hcl

CopyEdit

```
resource "aws_s3_bucket_website_configuration" "static_website" {
    bucket = aws_s3_bucket.static_website.id

    index_document {
        suffix = "index.html"
    }

    error_document {
        key = "error.html"
    }
}
```

```
}
```

 **Notes:**

- This **does not** upload or create HTML files. It only tells S3 **how to serve** them.
 - `suffix = "index.html"`: This is the default page when accessing the root.
 - `key = "error.html"`: This is served on 404s.
-

2. Create Local Website Files

Directory Structure:

```
go
CopyEdit
project-root/
|
└── build/
    ├── index.html
    └── error.html
```

index.html (Basic Template)

```
html
CopyEdit
<!DOCTYPE html>
<html>
<head>
    <title>My Static S3 Website</title>
    <meta name="description" content="My Static S3 Website">
    <meta name="keywords" content="terraform, s3, aws, hashicorp">
</head>
<body>
    <h1>Main Index HTML Page</h1>
</body>
</html>
```

error.html

```
html
CopyEdit
<!DOCTYPE html>
<html>
<head>
  <title>My Static S3 Website</title>
  <meta name="description" content="My Static S3 Website">
  <meta name="keywords" content="terraform, s3, aws, hashicorp">
</head>
<body>
  <h1>Oops! This page doesn't exist.</h1>
</body>
</html>
```

3. Manually Upload Files to S3

◆ Steps:

1. Go to S3 → Your Bucket.
 2. Click **Upload**.
 3. Select `index.html` and `error.html` from the `build/` folder.
 4. Confirm and Upload.
-

4. Accessing the Static Website

Steps:

1. Go to **Properties** of the S3 bucket.
2. Scroll to **Static Website Hosting**.
3. Copy the generated **website endpoint URL** (e.g. <http://your-bucket.s3-website-us-east-1.amazonaws.com>).
4. Open it in a browser.

Expected Behavior:

- `/` or no path → `index.html` is served.
 - `/non-existent.html` → `error.html` is served.
-

5. Shortcomings (and What to Improve Next)

Problem	Why it's Bad	What We'll Do Next
Files uploaded manually	Not reproducible. Can't work in CI/CD pipelines.	Automate with Terraform
Website URL not exposed via code	In a CI/CD environment, we can't manually fetch URLs from the AWS console.	Output URL in Terraform
Hardcoded structure	Hard to generalize for more projects or environments.	Parameterize later

Useful Terraform Commands Used

bash

CopyEdit

```
terraform fmt    # Auto-formats .tf files
terraform plan   # Previews changes to be made
terraform apply  # Applies the infrastructure
```

Key Learnings

Static Website Setup Requires:

- Bucket creation
- Public access settings
- Bucket policy for read access
- Website configuration

- HTML content

Manual Steps Are Suboptimal

Terraform should handle:

- Uploading the files.
- Outputting the site URL.

Terraform Powers

- Avoids hardcoding URLs (use `.arn` and other attributes).
- Handles infrastructure lifecycle.
- Uses `jsonencode()` to write inline JSON policies.

Closing Thoughts

 Even though we now have a functional static website, it's not production-grade until it's **fully automated**. Manual file uploads and console visits for URLs are anti-patterns in Infrastructure as Code.

Coming Up Next

In the **next lecture**, we will:

- Use Terraform to upload files to the bucket (no manual uploads).
- Use `terraform output` to expose the public URL automatically.
- Create a clean CI/CD-ready static hosting deployment.



Lecture 4: Automating Static Website Deployment on S3 with Terraform



Objective

In this lecture, we fully **automate** our S3 static website setup by:

1. Uploading website files (`index.html`, `error.html`) using Terraform.
 2. Exposing the static website endpoint as a Terraform output.
 3. Finalizing proper clean-up behavior using Terraform's dependency graph.
-



Part 1: Uploading Files Using `aws_s3_object` Resources



Why automate file upload?

Manual uploads are:

- Not repeatable
- Error-prone
- Infeasible in CI/CD

Terraform lets us manage **S3 objects** as infrastructure using the `aws_s3_object` resource.



Step-by-Step Terraform Code

hcl

CopyEdit

```
resource "aws_s3_object" "index_html" {  
  bucket      = aws_s3_bucket.static_website.id  
  key         = "index.html"  
  source      = "build/index.html"
```

```

    etag          = filemd5("build/index.html")
    content_type = "text/html"
}

resource "aws_s3_object" "error_html" {
  bucket      = aws_s3_bucket.static_website.id
  key         = "error.html"
  source      = "build/error.html"
  etag        = filemd5("build/error.html")
  content_type = "text/html"
}

```

Explanation

Field	Description
bucket	Links to the actual bucket using <code>.id</code>
key	Name of the object as it will appear in S3
source	Path to the file to be uploaded
etag	Optional; used to track changes (MD5 hash of the file)
content_type	MIME type; ensures browsers treat it as HTML (<code>text/html</code>)

 Without `content_type`, your browser might download the file instead of rendering it!

Terraform Gotcha: Duplicate Resource Labels

If you accidentally copy-paste without changing the resource label (e.g., use `index_html` twice), Terraform will error:

```

nginx
CopyEdit
Duplicate resource "aws_s3_object" configuration

```

 Reminder:

- **Resource address = <resource_type>.<resource_name>**

- Must be **unique** per module.
-

Part 2: Output the Static Website Endpoint

Manual fetching of the bucket URL from AWS Console is **non-ideal** in automated environments. Terraform lets us **expose outputs** to retrieve computed values.

outputs.tf File

Best practice is to separate outputs into their own file.

```
hcl
CopyEdit
output "static_website_endpoint" {
  value =
  aws_s3_bucket_website_configuration.static_website.website_endpoint
}
```

Accessing the Output via CLI

After applying:

```
bash
CopyEdit
terraform output static_website_endpoint
```

Returns:

```
CopyEdit
your-bucket.s3-website-us-east-1.amazonaws.com
```

Perfect for scripting or CI pipelines.

Applying the Changes

Commands Run:

```
bash
CopyEdit
terraform fmt
terraform apply
```

- Shows that only outputs changed (if infrastructure is already in place)
 - Creates two `aws_s3_object` resources if needed
 - Displays static website URL at the end
-

Testing the Website

Visit:

```
php-template
CopyEdit
http://<your-bucket>.s3-website-<region>.amazonaws.com/
```

You should see:

-  `index.html` on root path
 -  `error.html` when navigating to invalid paths
-

Cleanup and Destroy

```
bash
CopyEdit
terraform destroy
```

Smart Destruction Order

Terraform ensures:

1. `aws_s3_object` resources are destroyed **before** the bucket

2. Website configuration is removed
3. Bucket is deleted **last**

 You **can't delete non-empty buckets manually** via Terraform without doing this.

Key Insight

- Terraform automatically models **resource dependencies**.
 - Manually uploaded objects are **not tracked** → must be removed manually before `destroy`.
 - Managed objects are cleaned up **automatically!**
-

Final Key Concepts

Concept	Explanation
<code>aws_s3_object</code>	Manages individual files (objects) in S3
<code>filemd5()</code>	Computes hash for change detection
<code>content_type</code>	Ensures correct browser rendering
<code>terraform output</code>	Outputs usable runtime values like URLs
<code>terraform destroy</code>	Safely handles teardown, even with object dependencies
Output files (<code>outputs.tf</code>)	Separating outputs improves clarity and modularity

Real-World CI/CD Usage

When used in CI/CD pipelines:

- Terraform handles file uploads.
- Exposes the static site endpoint.
- Outputs can be passed to test jobs, frontend frameworks, DNS modules, etc.

Congratulations!

 You've now built a **fully automated S3 static website deployment pipeline** using Terraform.

You're managing:

- Website bucket lifecycle
- HTML file uploads
- Public access
- Static site routing
- Exposed endpoint for CI/CD

All reproducible, all codified, zero manual steps.

Optional Challenge

Try creating a **Terraform module** to reuse this static site deployment across multiple environments (e.g., `dev`, `prod`, `staging`). Consider parameterizing:

- Bucket name
- Source folder
- Content-type mapping

Data Sources



Terraform Lecture: Understanding Data Sources

(Extremely Detailed God-Level Notes – No Details Missed)



Overview: What are Data Sources in Terraform?

- **Purpose:**
Data sources in Terraform allow you to **query existing infrastructure** or **fetch information** from:
 - Cloud provider APIs (like AWS, GCP, etc.)
 - Other Terraform-managed infrastructure (even from other teams or projects)
- **Why It's Useful:**
You **don't always control** or want to manage every piece of infrastructure.
Sometimes, you just need to **reference an existing resource** without:
 - Recreating it
 - Importing it into your own Terraform project
 - Managing it



Real-World Use Case Example: Existing VPC and Role

Imagine this scenario:

Resource	Managed By You?	Action Needed
----------	-----------------	---------------

Existing VPC ✗ Another team Use it in your EC2 / RDS

Existing IAM Role ✗ Another team Use in your application

-

Your Project's Needs:

- You need to launch an **EC2 instance** and an **RDS database**.
- Both need to be associated with an existing **VPC** and an **IAM role**.
- But you **don't own** or manage these shared resources.

✗ What You Should NOT Do

- **Do not re-create the existing VPC / Role** using **resource** blocks.
- **Do not import the VPC / Role** into your state if you're not responsible for managing it — this would:
 - Bring the resource under your management
 - Introduce potential conflicts if the owning team changes it

✓ What You SHOULD Do: Use Data Sources

Terraform provides a way to reference such resources via **data** blocks.

✗ Basic Syntax:

hcl

CopyEdit

```
data "aws_vpc" "existing" {  
  
  filter {  
  
    name    = "tag:Name"  
  
    values = ["shared-vpc"]  
  
  }  
  
}
```

- `data "aws_vpc" "existing"` → Refers to an **existing VPC**
- `filter` block tells Terraform **how to find it** via AWS APIs

You can now use the VPC like this:

```
hcl  
CopyEdit  
subnet_id = data.aws_vpc.existing.default_route_table_id  
  
•
```

What Terraform Does Behind the Scenes

1. **Reads the `data` block**
2. **Sends API calls** to AWS (or any other provider)
3. **Finds the matching resource** using filters
4. **Makes it available** to use within other resources

Other Example: Existing IAM Role

hcl

[CopyEdit](#)

```
data "aws_iam_role" "existing" {  
  
  name = "my-app-role"  
  
}
```

You can now use this role in your EC2 or Lambda setup, without ever managing or modifying it yourself.

Common Use Cases for Data Sources

Use Case	Description
Referencing shared infrastructure	Like VPCs, IAM roles, subnets, AMIs
Multi-team collaboration	Your team only references infrastructure owned by another team
External data	Example: Get a latest AMI ID dynamically

Fetch provider-specific metadata	Like availability zones, regions, etc.
----------------------------------	--



Important Note

- Data sources are **read-only**.
You can reference them, but **cannot modify** them.
 - Still subject to the **lifecycle of the actual resource**.
If the other team deletes or renames the VPC, your code might break.
 - You **must provide accurate filters**; otherwise, Terraform will **fail to find** the resource and throw an error.
-



How **data** Complements **resource**

resource Block

Defines and manages new infra

Adds to Terraform state

You own it

data Block

Reads existing infra

Doesn't manage the state

You just use it



Practical Preview

- Future lectures will include **practical demos**:
 - Using **data** sources to fetch:
 - AMIs
 - VPCs
 - IAM roles
 - Availability Zones
 - You will see how **data** blocks integrate seamlessly into the rest of your infrastructure definitions.
-

Summary (TL;DR)

- Data sources help **reference existing infrastructure**.
 - Great for **shared resources, read-only usage**, and **team isolation**.
 - You use **data** blocks with **filters or names** to retrieve what you need.
 - You can then plug this data into your resource definitions (like EC2, RDS, etc.).
 - Terraform handles fetching this data via **API calls** behind the scenes.
-

Sample Code Blocks (Ready to Use)

Example 1: Use an existing VPC

h

CopyEdit

```
data "aws_vpc" "existing_vpc" {
```

```
filter {

    name    = "tag:Name"

    values = ["production-vpc"]

}

}
```

Example 2: Use an existing IAM role

h

CopyEdit

```
data "aws_iam_role" "existing_role" {

    name = "shared-app-role"

}
```

Example 3: Fetch latest Ubuntu AMI dynamically

hcl

CopyEdit

```
data "aws_ami" "ubuntu" {

    most_recent = true
```

```
filter {

    name     = "name"

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

}

owners = [ "099720109477" ] # Canonical

}
```



Terraform Lecture Notes – Multi-Region AMI Fetch & EC2 Instance Provisioning

◆ Overview

You are using Terraform to:

1. **Fetch Ubuntu AMIs from two AWS regions** (default & us-east-1).
2. **Output those AMI IDs** for inspection.
3. **Provision an EC2 instance** using a hardcoded AMI ID.

You faced the following errors:

- `MissingInput`: No subnets found for the default VPC
 - `Provider configuration not present`
 - AMI for both regions appeared to be **the same**.
 - You mentioned **you previously deleted default subnets**.
-



1. Terraform Files – Clean, Complete Code (With Fixes)

main.tf

```
hcl
CopyEdit
terraform {
  required_version = ">= 1.3.0"

  required_providers {
    aws = {

```

```

        source  = "hashicorp/aws"
        version = "~>5.0"
    }
}
}

provider "aws" {
    region = "eu-west-1"
}

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

```

ami.tf

hcl
CopyEdit

```

data "aws_ami" "ubuntu_eu" {
    most_recent = true
    owners      = ["099720109477"] # Canonical (Ubuntu)

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

    filter {
        name    = "virtualization-type"
        values = ["hvm"]
    }
}

data "aws_ami" "ubuntu_us" {
    provider    = aws.us-east
    most_recent = true
    owners      = ["099720109477"]

    filter {

```

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

filter {
    name    = "virtualization-type"
    values = ["hvm"]
}
}
```

outputs.tf

```
hcl
CopyEdit
output "ubuntu_ami_data_eu" {
    value = data.aws_ami.ubuntu_eu.id
}

output "ubuntu_ami_data_us" {
    value = data.aws_ami.ubuntu_us.id
}
```

compute.tf

```
hcl
CopyEdit
resource "aws_instance" "web" {
    ami          = data.aws_ami.ubuntu_eu.id
    instance_type = "t2.micro"

    # Must provide subnet for non-deprecated instance launch
    subnet_id = aws_subnet.default.id

    associate_public_ip_address = true

    root_block_device {
        volume_type      = "gp3"
        volume_size      = 10
        delete_on_termination = true
    }
}
```

```
}

tags = {
    Name = "MyWebInstance"
}
}
```

network.tf (for recreated subnet)

```
hcl
CopyEdit
resource "aws_vpc" "default" {
    cidr_block = "10.0.0.0/16"
    enable_dns_support = true
    enable_dns_hostnames = true

    tags = {
        Name = "Recreated Default VPC"
    }
}

resource "aws_subnet" "default" {
    vpc_id      = aws_vpc.default.id
    cidr_block = "10.0.1.0/24"
    availability_zone = "eu-west-1a"

    tags = {
        Name = "Recreated Subnet"
    }
}
```

What Is Happening Step by Step

1. Terraform Initialization & Provider Setup

- You are using **two providers**:

- Default (EU region, no alias).
- `us-east` (aliased provider).

 **Issue fixed:** You initially forgot to add `alias = "us-east"` when referencing the second provider, causing the error:

```
nginx
CopyEdit
Provider configuration not present
```

● 2. AMI Fetching Logic

- Both AMIs are fetched from **Canonical (099720109477)**, for **Ubuntu 20.04**.
- Filter ensures:
 - `hvm` virtualization type.
 - Only `ubuntu-focal` matching images are picked.
 - Most recent is selected.

 **Why you saw same AMI IDs for both regions:**

- Canonical re-uses the same image names across regions.
- But **AMI IDs are region-specific**. If they matched, you were likely hitting:
 - **Default region provider** for both calls (due to missing alias).
 - Now fixed with `provider = aws.us-east`.

 Run `terraform output` to confirm both AMI IDs differ.

● 3. Error: MissingInput – No Subnets Found

```
pgsql
CopyEdit
MissingInput: No subnets found for the default VPC
```



Root Cause:

You **deleted all subnets** tied to the default VPC.

Terraform cannot launch EC2 without knowing where to place the instance.



Fix: Recreate Subnet + Explicitly Provide `subnet_id`

To fix this:

1. Recreate a subnet (`aws_subnet.default`) as shown in `network.tf`.
2. Assign its `id` explicitly to your EC2 resource via `subnet_id = aws_subnet.default.id`.

This satisfies the placement requirement.

Full Working Flow Recap

Step-by-step:

Step	Description
1	Provider block defines both <code>eu-west-1</code> and <code>us-east-1</code>
2	AMI data sources pull latest Ubuntu AMI from both regions
3	Outputs print AMI IDs (useful for inspection or reference)
4	EC2 instance is provisioned with EU AMI and uses recreated subnet
5	Subnet is created manually since default ones were deleted
6	Errors resolved by explicitly adding <code>provider</code> and <code>subnet_id</code>



Testing Suggestions

- Run `terraform apply` and check AMI IDs in output.
- SSH into the instance to verify connectivity (if public IP attached).

- Optionally add `key_name` to use your SSH key.
-



Final Tips

- Don't rely on "default" VPC/subnet unless untouched.
 - Use `provider = aws.us-east` explicitly when referencing aliased provider.
 - Always use `most_recent = true` when using wildcards for AMI search.
 - Add `depends_on` between resources if Terraform doesn't auto-resolve order.
-



Outputs Example (After Fix)

```
ini
CopyEdit
ubuntu_ami_data_eu = "ami-0abcdef1234567890"
ubuntu_ami_data_us = "ami-0789abcdef1234560"
```

Would you like me to now break this down **line by line**, or go into **execution and debugging** in your console step by step?



Terraform Lecture 3: More on Data Sources

Title: Using `aws_caller_identity` and `aws_region` Data Sources

"Read, don't recreate. Reference, don't manage."

Learning Objectives:

By the end of this lecture, you will:

- Understand how to use built-in data sources like `aws_caller_identity` and `aws_region`
 - Learn how to dynamically retrieve the AWS Account ID and current region
 - Understand **meta-arguments** like `provider` and how they apply to data sources
 - Know how to query **another region's** metadata using **aliased providers**
 - Format and clean up Terraform code using `terraform fmt`
-



Why Do We Need These Data Sources?

? Problem 1: You need the AWS Account ID or the ARN of the caller

- Useful for dynamically building IAM policies
- You should not hardcode the account ID (anti-pattern)

? Problem 2: You need to reference the current AWS region

- Maybe for setting metadata or naming resources

- Again, **don't hardcode it** — read it dynamically
-



DATA SOURCE 1: `aws_caller_identity`

Purpose:

Fetches the **identity of the current AWS caller** (user, account, and ARN)

Terraform Official Doc:

[aws_caller_identity – Terraform Registry](#)

Code:

```
hcl
CopyEdit
data "aws_caller_identity" "current" {}
```

Output Example:

```
hcl
CopyEdit
output "caller_identity_info" {
  value = data.aws_caller_identity.current
}
```

Returned Attributes:

- `account_id` → AWS account number (e.g., `123456789012`)
- `user_id` → Unique identifier for IAM user or assumed role
- `arn` → Amazon Resource Name (e.g.,
`arn:aws:iam::123456789012:user/terraform-user`)

Use Case:

```
hcl
CopyEdit
resource "aws_iam_policy" "example" {
  name      = "ExamplePolicy"
```

```
description = "A sample policy using dynamic Account ID"

policy = jsonencode({
  Version = "2012-10-17"
  Statement = [
    {
      Effect   = "Allow"
      Action   = "s3>ListBucket"
      Resource =
"arn:aws:s3:::mybucket-${data.aws_caller_identity.current.account_id}"
    }
  ]
})
}
```



DATA SOURCE 2: `aws_region`



Returns metadata about the **current region** used by the AWS provider



Terraform Official Doc:

[aws_region – Terraform Registry](#)



Code:

hcl
CopyEdit
`data "aws_region" "current" {}`



Output Example:

hcl
CopyEdit
`output "current_region" {
 value = data.aws_region.current.name
}`



Returned Attributes:

- `name` → Current region (e.g., "us-east-1")
- `description` → Full region description (e.g., "US East (N. Virginia)")
- `endpoint` → Regional endpoint for EC2 (e.g., "ec2.us-east-1.amazonaws.com")

Use Case:

```
hcl
CopyEdit
resource "aws_s3_bucket" "mybucket" {
  bucket = "my-bucket-${data.aws_region.current.name}"
}
```

Querying Information from Another Region

Goal:

We want to fetch region data **from a different AWS region** (not the one our provider is using)

Solution: Use a provider alias

Step-by-step:

1. Define a second provider with an alias:

```
hcl
CopyEdit
provider "aws" {
  alias  = "us_east"
  region = "us-east-1"
}
```

2. Use this provider in your data source:

```
hcl
CopyEdit
data "aws_region" "us_east_info" {
```

```
    provider = aws.us_east
}
```

3. Output it:

```
hcl
CopyEdit
output "us_east_region" {
  value = data.aws_region.us_east_info.name
}
```

🛑 Important Terraform Concept: Meta-Arguments

Argument	Type	Meaning
provider	Meta	Lets you specify which provider alias to use
depends_on	Meta	Ensures one block waits on another
count/for_each	Meta	Controls repetition logic

Meta-arguments are supported in both `resource` and `data` blocks

→ Even though not listed in provider documentation.

🧹 Cleanup: `terraform fmt`

After modifying your files:

```
bash
CopyEdit
terraform fmt
```

- ✅ Standardizes code formatting
- ✅ Aligns indentation
- ✅ Makes code clean and readable



FULL CODE SNIPPET (Copy-Paste Ready)

h
CopyEdit

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

data "aws_caller_identity" "current" {}

data "aws_region" "current" {}

output "caller_identity" {
  value = data.aws_caller_identity.current
}

output "aws_region_name" {
  value = data.aws_region.current.name
}
```



OPTIONAL: Fetch from Another Region

hcl
CopyEdit

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

data "aws_region" "us_east_info" {
  provider = aws.us_east
}

output "us_east_region_name" {
  value = data.aws_region.us_east_info.name
}
```

Summary

Feature	<code>aws_caller_identity</code>	<code>aws_region</code>
What it does	Fetches AWS account/user info	Fetches current region details
Use case	Dynamic IAM policy ARNs	Region-based naming or logic
Returns	<code>account_id, arn, user_id</code>	<code>name, description, endpoint</code>
Editable?	 Read-only	 Read-only
Provider override?	 Yes, via <code>provider = ...</code>	 Yes

Lecture 4: Fetching VPC Information with Terraform Data Sources

Goal of This Lecture

Use Terraform **data sources** to **fetch an existing VPC** (created manually or by another team) using **tags** — *without managing that VPC directly*.

Why This Matters

In real-world infrastructure, not everything is managed by a single Terraform project.

- Platform/Networking teams may create **shared VPCs**.
- You might be deploying services **into** these VPCs.
- You **should not** hardcode VPC IDs — it's brittle and error-prone.

 Instead: Use the `aws_vpc` **data source** to dynamically retrieve VPC info by tag.

Pre-requisites

Manual VPC setup in AWS Console:

Create a VPC via the AWS Console:

- Name: `console-managed`
- CIDR block: `10.0.0.0/16`
- Tag: `env = prod`

 Example Tag Table:

Key	Value
name	console-manage
env	d
env	prod

Terraform Data Source: aws_vpc

Doc:

[terraform.io - aws_vpc data source](#)

Purpose:

Reads info about **existing VPCs** (not managed by this Terraform project)

FULL CODE

```
hcl
CopyEdit
provider "aws" {
  region = "us-east-1" # or your chosen region
}

# 🔎 Fetch VPC by tag
data "aws_vpc" "prod_vpc" {
  tags = {
    env = "prod"
  }
}

# 📦 Output VPC ID
output "prod_vpc_id" {
  value = data.aws_vpc.prod_vpc.id
}
```

Run Commands:

bash
CopyEdit

```
terraform init  
terraform plan
```

Expected Output:

```
hcl  
CopyEdit  
prod_vpc_id = "vpc-0f7abcd1234567890"
```

This VPC ID should **match** the one shown in the AWS Console (VPC section).

How aws_vpc Works

Required:

- No need to pass the VPC ID
- Uses **filters** like `tags` or `cidr_block`

Available Attributes:

- `id` → VPC ID
 - `cidr_block`
 - `default` → Whether this is the default VPC
 - `state`
 - `tags`
 - `owner_id`
 - ...
-

Error Handling: What if No Match?

Scenario:

```
hcl
CopyEdit
tags = {
  env = "nonexistent"
}
```

📝 Result:

```
bash
CopyEdit
| Error: no matching VPC found
```

Terraform will **fail fast** if no resource matches your filter.

⚠ Pitfall: Matching the Wrong VPC

Let's say you mistakenly wrote:

```
h
CopyEdit
tags = { env = "prod" }
```

But your intention was:

```
hcl
CopyEdit
tags = { env = "non-prod" }
```

🚫 The VPC **still exists** → Terraform will match it
But it's the **wrong environment!**

This could lead to:

- ✗ Deploying dev code to prod
- ✗ Security violations
- ✗ Overwrites or cost leaks

✓ Solution Preview (covered in later lectures):

- Add **validation conditions**
 - Implement **naming conventions**
 - Use `count`, `length()`, or custom modules for safety
-

Summary Table

Concept	Details
Data Source	<code>aws_vpc</code>
Purpose	Fetch existing VPCs (not managed by Terraform)
Filter Used	<code>tags = { env = "prod" }</code>
Output Attribute	<code>.id</code> — the VPC ID
When to Use	Reusing shared or externally-managed networking infrastructure
Safe Practice	Avoid hardcoding; always filter and validate
Danger Zone 	Wrong tag = wrong VPC matched = potential misdeployment

Test Yourself (Mini Exercise)

Q: How would you fetch a **non-default** VPC with CIDR block `10.0.0.0/16`?

```
hcl
CopyEdit
data "aws_vpc" "custom_vpc" {
  filter {
    name    = "cidr-block"
    values  = ["10.0.0.0/16"]
  }

  filter {
    name    = "isDefault"
    values  = ["false"]
  }
}
```



Clean-up Commands

```
bash
CopyEdit
terraform fmt
terraform validate
terraform plan
```



Pro Tips

- Always double-check your **filters** — a wrong tag can silently match wrong resources
- Combine **tags** with other filters (e.g., CIDR) for **stronger matches**
- Use **terraform console** to inspect values:

```
bash
CopyEdit
terraform console
> data.aws_vpc.prod_vpc.cidr_block
```



Real-World Use Cases

- Reference shared VPCs created by networking/platform teams
- Create subnets, route tables, or EC2 instances **inside** a pre-existing VPC
- Use centralized IAM/Networking setup without duplicating infrastructure logic

Lecture 5: Fetching AWS Availability Zones Dynamically via Terraform Data Sources

Goal

Use a **Terraform data source** to dynamically fetch a list of **available Availability Zones (AZs)** in the current region — *instead of hardcoding them.*

Why It Matters

Benefit

No Hardcoding

Availability Zones vary by region (e.g., `us-east-1a`, `eu-west-1a`, etc.)

Region-Agnostic Code

Avoid brittle references tied to one AWS region

Resilience

Skip zones in `impaired` or `unavailable` state

Scaling

Used when creating N subnets, ASGs, etc., across multiple AZs

Explanation

Full Code: `main.tf`

hcl

CopyEdit

```
provider "aws" {  
    region = "eu-west-1" # Replace with your region  
}
```

```
#  Fetch available AZs in the region  
data "aws_availability_zones" "available" {  
    state = "available" # Only fetch healthy AZs  
}
```

```
#  Output AZ names (like eu-west-1a, eu-west-1b)
```

```
output "availability_zone_names" {
  value = data.aws_availability_zones.available.names
}
```

Run the Code

bash
CopyEdit
`terraform init`
`terraform plan`

Sample Output

hcl
CopyEdit
`availability_zone_names = [`
 `"eu-west-1a",`
 `"eu-west-1b",`
 `"eu-west-1c"`
`]`

Understanding the Data Source: `aws_availability_zones`

Official docs: Terraform Registry – `aws_availability_zones`

◆ What it fetches:

- A list of AZs in the current region that are available to the current AWS account.

◆ Supported Filters:

- `state = "available"`  Recommended
- `all_availability_zones = true` (optional, includes even impaired/unavailable zones)

Object Structure Returned:

```
hcl
CopyEdit
{
  id      = "eu-west-1"    # Region
  names   = [ "eu-west-1a", "eu-west-1b", "eu-west-1c" ]  # AZ names
  zone_ids = [ "euw1-az2", "euw1-az1", "euw1-az3" ]
}
```



Deep Dive: `names`, `zone_ids`, and `id`

Property	Type	Example	Meaning
names	list	["eu-west-1a", ...]	Human-readable AZ identifiers
zone_id	list	["euw1-az2", ...]	Stable internal zone IDs (used in some services)
id	string	"eu-west-1"	Region from which data was fetched



⚠ Gotcha: Split Expressions — Misunderstanding Fixed

✗ Initial (incorrect assumption):

```
h
CopyEdit
output "az_ids" {
  value = data.aws_availability_zones.available[*].id
}
```

This **does not work** — `aws_availability_zones` is a **single object**, not a list of objects.

✓ Correct:

```
hcl
CopyEdit
output "availability_zone_names" {
  value = data.aws_availability_zones.available.names
}
```



Bonus: Access Individual AZs

hcl
CopyEdit

```
output "first_az" {
    value = data.aws_availability_zones.available.names[0]
}

output "third_zone_id" {
    value = data.aws_availability_zones.available.zone_ids[2]
}
```



Real-World Use Case Example: Creating One Subnet Per AZ

h

CopyEdit

```
resource "aws_subnet" "public" {
    count          =
    length(data.aws_availability_zones.available.names)
    vpc_id         = var.vpc_id
    cidr_block     = cidrsubnet(var.vpc_cidr_block, 8, count.index)
    availability_zone =
    data.aws_availability_zones.available.names[count.index]
}
```



Testing the Data Source

bash

CopyEdit

```
terraform console
> data.aws_availability_zones.available
{
    "id" = "eu-west-1"
    "names" = tolist([
        "eu-west-1a",
        "eu-west-1b",
        "eu-west-1c",
```

```
    ])
    "zone_ids" = tolist([
        "euw1-az1",
        "euw1-az2",
        "euw1-az3",
    ])
}
```

Best Practices

Best Practice	Why it Matters
Use <code>state = "available"</code>	Excludes impaired AZs
Avoid Hardcoding AZ Names	Names differ across regions/accounts
Use <code>count</code> or <code>for_each</code> with AZs	To deploy subnets/load balancers across zones
Inspect <code>zone_ids</code> if needed	Required for services like Local Zones or Outposts

Clean Up & Format

bash
CopyEdit
`terraform fmt`
`terraform validate`
`terraform plan`

Recap of Key Concepts

Concept	Details
Data Source	<code>aws_availability_zones</code>
Returns	Object with <code>names</code> , <code>zone_ids</code> , and <code>id</code>
Use Case	Region-agnostic infrastructure deployments
Output Format	List of names (e.g., ["us-east-1a", "us-east-1b"])

Common Mistake Treating the data source like a list of objects (it's a single object)

Safety Best Practice Always filter with `state = "available"`

You've Now Mastered:

- Dynamic AZ discovery
- Terraform data object traversal
- Region-safe infra patterns
- Avoiding common output errors

Lecture 6: Defining IAM Policy Documents Using Terraform Data Sources

Goal of This Lecture

Use the `aws_iam_policy_document` data source in Terraform to programmatically generate an AWS IAM policy document — without manually writing raw JSON.

Why This Matters

Benefit

Avoids manual JSON

Define IAM policies in HCL (HashiCorp Configuration Language)

More modular and reusable

Reuse across multiple modules/resources

Supports validation

Invalid actions/principals will error at plan time

More readable and Terraform-native

Explanation

Use Case: S3 Public Read Access Policy for Static Website

We want to allow **public read access** (`s3:GetObject`) to **all objects** in an S3 bucket.

Full Terraform Code: `main.tf`

```
hcl
CopyEdit
provider "aws" {
  region = "eu-west-1" # or your preferred region
}
```

```

# ① Create the S3 bucket
resource "aws_s3_bucket" "public_read_bucket" {
  bucket = "my-public-read-bucket-123456"
}

# ② Define the IAM policy document using data source
data "aws_iam_policy_document" "static_website" {
  statement {
    sid = "PublicReadGetObject"

    principals {
      type     = "*"
      identifiers = ["*"]
    }

    actions   = [ "s3:GetObject" ]
    resources = [ "${aws_s3_bucket.public_read_bucket.arn}/*" ]
  }
}

# ③ Output the generated policy as JSON
output "iam_policy_json" {
  value = data.aws_iam_policy_document.static_website.json
}

```



Breakdown of Each Block

- ◆ **aws_s3_bucket.public_read_bucket**

↳ [CopyEdit](#)

```
resource "aws_s3_bucket" "public_read_bucket" {
  bucket = "my-public-read-bucket-123456"
}
```

- Creates an S3 bucket
 - Name must be globally unique
-

- ◆ **data.aws_iam_policy_document.static_website**

```
hcl
CopyEdit
data "aws_iam_policy_document" "static_website" {
  statement {
    sid = "PublicReadGetObject"

    principals {
      type      = "*"
      identifiers = ["*"]
    }

    actions   = [ "s3:GetObject" ]
    resources = [ "${aws_s3_bucket.public_read_bucket.arn}/*" ]
  }
}
```

Explanation

Field	Value	Meaning
statement	block	Each statement maps to a JSON policy array entry
sid	"PublicReadGetObject"	Statement ID — optional but useful
principals	type = "*"	All principal types (e.g., AWS, Service, etc.)
	identifiers = ["*"]	Allow any identity — i.e., public access
actions	["s3:GetObject"]	Allow reading objects
resources	"bucket_arn/*"	Applies to all objects in the bucket

- ◆ **output.iam_policy_json**

```
hcl
CopyEdit
output "iam_policy_json" {
  value = data.aws_iam_policy_document.static_website.json
}
```

- Returns the entire **computed IAM policy** as a valid JSON string
 - This can be directly attached to a bucket policy or IAM policy resource
-



Terraform Plan Behavior

Case 1: With dynamic values (e.g., using `${bucket.arn}`)

`terraform plan` output:

```
ini
CopyEdit
iam_policy_json = (known after apply)
```

- ➤ Because the bucket ARN isn't known until the resource is created
-

Case 2: With hardcoded resource string

Change:

```
hcl
CopyEdit
resources = [ "arn:aws:s3:::my-static-site/*" ]
```

Then plan:

```
bash
CopyEdit
terraform plan
```

- `terraform plan` now shows full JSON
 - Why? ➤ Because no interpolation needed → value is static → resolved at **plan time**
-

Sample Output (Pretty Printed JSON)

```
json
CopyEdit
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::my-public-read-bucket-123456/*"
    }
  ]
}
```

Terraform JSON Conversion: Behind the Scenes

Terraform auto-converts the HCL statement into a valid JSON IAM policy via the `json` attribute.

Internally, it's like using:

```
hcl
CopyEdit
jsonencode({
  Version = "2012-10-17"
  Statement = [
    {
      Sid      = "PublicReadGetObject"
      Effect   = "Allow"
      Action   = "s3:GetObject"
      Principal = "*"
      Resource = "arn:aws:s3:::bucket-name/*"
    }
  ]
})
```

But you don't need to handle formatting or escaping — Terraform handles all of it.

Advanced Use: Multiple Statements

hcl
CopyEdit

```
data "aws_iam_policy_document" "multi_statement" {
  statement {
    sid = "ReadAccess"
    actions = ["s3:GetObject"]
    resources = ["arn:aws:s3:::mybucket/*"]
    principals {
      type      = "*"
      identifiers = ["*"]
    }
  }

  statement {
    sid = "WriteAccess"
    actions = ["s3:PutObject"]
    resources = ["arn:aws:s3:::mybucket/uploads/*"]
    principals {
      type      = "AWS"
      identifiers = ["arn:aws:iam::123456789012:user/uploader"]
    }
  }
}
```

Validation Benefits

If you make a mistake like this:

hcl
CopyEdit

```
principals {
  type = "*"
  # Forgot identifiers
}
```

Terraform will error:

swift
CopyEdit
The argument "identifiers" is required.

💡 This prevents subtle bugs that raw JSON files won't catch until AWS rejects them.

🔁 Alternatives to This Method

Method	Pros	Cons
<code>aws_iam_policy_document</code> (✓ THIS)	Validated, modular, Terraform-native	Requires Terraform knowledge
Raw JSON string in-line	Quick for tiny policies	Easy to make syntax errors
External <code>.json</code> files + <code>file()</code>	Reusable across tools	No validation; hard to debug
Locals with <code>jsonencode()</code>	Easy composition in HCL	Not validated like the data source

✓ Best Practices

Practice	Reason
Use the <code>json</code> output	Always attach to <code>policy</code> field of IAM resources
Use <code>principals</code> correctly	Must always have both <code>type</code> and <code>identifiers</code>
Use multiple <code>statement</code> blocks	For logically distinct access rules
Avoid using <code>*</code> in prod	Too permissive — use for learning only
Validate via <code>terraform plan</code>	Catch issues before apply

]-\$ Format & Apply

bash

CopyEdit

```
terraform fmt  
terraform validate  
terraform plan  
terraform apply
```



Recap: What You Learned

Concept	Details
Data source: <code>aws_iam_policy_document</code>	Build IAM policies from HCL
<code>statement</code> block	Maps to a policy statement
<code>principals, actions, resources</code>	Define who, what, and where
<code>json</code> attribute	Outputs a valid IAM policy in JSON
Validated syntax	Terraform checks for required fields
Use cases	S3 bucket policies, IAM inline policies, etc.



You Can Now:

- Build validated IAM policies in Terraform
- Avoid raw JSON and syntax issues
- Dynamically attach IAM policies using data sources
- Reuse and scale policy definitions across projects

Input variables,locals and outputs



Lecture: Terraform Input Variables – Theory & Usage

🎯 Goal of This Lecture

To understand what input variables are in Terraform, why they are useful, how to define and use them, and how variable values are passed into the configuration using different methods, along with precedence rules.

🧠 What Are Input Variables?

Terraform **input variables** are **parameters** that allow users to **customize configurations without editing the source code**.

💡 Why Use Them?

- Make code **reusable**, **composable**, and **modular**
 - Avoid hardcoding values (which we've done so far using `locals` or directly in `resource` blocks)
 - **Expose configuration options** to the end-user (DevOps engineers, infra teams)
 - Encourage **declarative best practices**
-

📁 Where to Define Input Variables

 Conventionally, input variables are defined in a dedicated file:

hcl

CopyEdit

 Recommended file name

`variables.tf`

You can technically define variables in any `.tf` file, but `variables.tf` improves maintainability.

Syntax for Declaring a Variable

hcl

CopyEdit

```
variable "instance_type" {  
    type      = string  
    description = "EC2 instance type"  
    default    = "t2.micro"  
}
```

Explanation:

Field	Purpose
<code>type</code>	Enforces allowed input (string, number, bool, list, map, etc.)
<code>description</code>	Helpful docstring for future readers or teams
<code>on</code>	
<code>default</code>	If set, makes variable optional – Terraform won't prompt at runtime
<code>sensitive</code>	If <code>true</code> , Terraform hides the value in CLI output (but not in state)
<code>validation</code>	Allows conditions like <code>length > 3</code> , regex, or custom rule enforcement

Internal Terraform Behavior – Sensitive Values

hcl

CopyEdit

```
variable "db_password" {  
    description = "Database root password"  
    type        = string  
    sensitive   = true
```

```
}
```

Terraform Behavior:

- Terraform **hides sensitive values** in `terraform plan` and `terraform apply` output
 - But **⚠ sensitive values are still stored in plaintext in `terraform.tfstate`**, unless you **encrypt the state backend** (e.g., use S3 + KMS + DynamoDB)
-

Using Variables in Your Code

hcl

CopyEdit

```
resource "aws_instance" "example" {  
    instance_type = var.instance_type  
}
```

Variables are accessed using `var.<variable_name>`.

Ways to Provide Variable Values

1. Defaults inside `variables.tf`

hcl

CopyEdit

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

If no value is passed explicitly, the **default is used**.

2. Environment Variables

```
bash
CopyEdit
export TF_VAR_region="us-east-1"
```

- Must be prefixed with `TF_VAR_`
 - Works well for automation and CI/CD pipelines
 - Supports all variable types (JSON for complex types)
-

3. Variable Definition Files

terraform.tfvars

hcl

CopyEdit

```
region = "us-east-2"
instance_type = "t3.medium"
```

terraform.tfvars.json

json

CopyEdit

```
{
  "region": "us-east-2",
  "instance_type": "t3.medium"
}
```

 Terraform will automatically load:

- `terraform.tfvars`
- `terraform.tfvars.json`

You can also specify **custom varfiles** using the CLI (see below).

4. Command Line Flags

bash

CopyEdit

```
terraform apply -var="region=us-west-1"  
terraform apply -var-file="prod.tfvars"
```

- `-var` = Inline variable definition
 - `-var-file` = Load a specific file
-

Variable Precedence – Terraform's Loading Order

plaintext

CopyEdit

 Lowest precedence

1. Default value in `variables.tf`
 2. `terraform.tfvars` / `terraform.tfvars.json`
 3. Environment variables (`TF_VAR_`)
 4. `-var-file` CLI flag
 5. `-var` CLI flag
-

 Highest precedence

 Higher precedence overrides the lower. If the same variable is defined in multiple places, the **last loaded value wins**.

Terraform CLI Behavior with Variables

Case 1: Variable With No Default and No Supplied Value

hcl

CopyEdit

```
variable "region" {
```

```
    type = string  
}  
  
Running:
```

```
bash  
CopyEdit  
terraform apply
```

→ Terraform will prompt:

```
css  
CopyEdit  
var.region  
Enter a value:
```

Case 2: Variable With Default

```
hcl  
CopyEdit  
variable "region" {  
  default = "us-east-1"  
}
```

→ Terraform will not ask for the value if no other override is provided.

⚠ Common Mistakes and Gotchas

✗ Mistake

Forgetting to pass required variables with no default

✓ Correct

Use `-var` or `-var-file`

Thinking `sensitive = true` encrypts state

It only hides CLI output

Using wrong `TF_VAR_` syntax

Must match variable name exactly

Expecting <code>.tfvars</code> to load automatically if not named properly	Only <code>terraform.tfvars</code> auto-loads
Overriding default accidentally with env var	Be aware of your shell's exports
Defining a complex object using <code>-var</code>	Use <code>-var-file</code> or JSON for readability

Behind the Scenes – Terraform Internals

When Terraform Resolves Variable Values

- Variables are resolved **before the plan phase**, during the **init/configuration loading** phase
 - **Interpolation** like `${var.region}` happens at *render time*, before building the graph
 - Terraform **builds a dependency graph**, so if a variable influences a `resource`, it will trigger updates if the value changes
 - Even if a variable is not used in a `resource`, its value still ends up in `terraform.tfstate` if interpolated or stored anywhere (like in outputs)
-

Sample Variable Declaration + Usage

hcl

CopyEdit

```
# variables.tf
variable "aws_region" {
    type      = string
    description = "The AWS region to deploy to"
    default    = "us-east-1"
}
```

```
# main.tf
```

```
provider "aws" {
```

```
    region = var.aws_region  
}
```

CLI Output (with no override):

```
bash  
CopyEdit  
$ terraform plan  
provider.aws: version = "~> 4.0"  
region = "us-east-1"
```



Summary of Concepts Covered

Concept	Description
Input Variables	User-defined inputs for customization
Declaration	Use <code>variable</code> block, with type, description, default, sensitive
Access	<code>var.<name></code>
Value Sources	Default, env vars, <code>.tfvars</code> , CLI
Precedence	CLI overrides all others
CLI Behavior	Prompts for required vars, uses defaults if present
Internals	Evaluation before graph, stored in state, affects planning



Next Step

In the next lecture, you'll **implement variables in practice**, replacing hardcoded values with `var.*`, supplying `.tfvars`, and experimenting with overrides.

Lecture 2: Using Input Variables to Configure AWS Region — Gotchas, CLI Behavior, and State Risks

GOAL

Implement input variables to make the AWS region configurable and understand the **risks** of using variables to configure the **provider block**.

 This is **not just a feature lesson** — this is a **warning** about how Terraform behaves when your provider configuration becomes dynamic.

Project Structure

```
bash
CopyEdit
08-input-vars-locals-outputs/
|
└── provider.tf      # Terraform & AWS provider block
└── compute.tf        # EC2 instance configuration
└── variables.tf      # Variable declarations
```

Step 1: Create Project Directory

```
bash
CopyEdit
mkdir 08-input-vars-locals-outputs
cd 08-input-vars-locals-outputs
```

Step 2: provider.tf

```
hcl
CopyEdit
terraform {
  required_version = ">= 1.7.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
```

```
        version = ">= 5.0, < 6.0"
    }
}
}

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

Explanation

- `required_version` ensures compatibility
 - `required_providers` pins AWS provider version between 5.x and <6.x
 - `region = var.aws_region` is the key customization via input variable
-

Step 3: variables.tf

hcl
CopyEdit

```
variable "aws_region" {
  description = "The AWS region to deploy into"
  type        = string
}
```

 **Why type matters:** It enables validation and prevents accidental misconfiguration.

Behavior: What Happens When You Don't Set a Value?

bash
CopyEdit

```
terraform apply
```

CLI Prompt:

```
text
CopyEdit
var.aws_region
  The AWS region to deploy into
  Enter a value:
```

 **Good:** Terraform prompts for missing variables

Step 4: compute.tf

```
hcl
CopyEdit
resource "aws_instance" "compute" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  root_block_device {
    delete_on_termination = true
    volume_size          = 10
    volume_type           = "gp3"
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true

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

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```

Terraform CLI Behavior Exploration

Test Case 1: Set Region via Interactive Prompt

bash

CopyEdit

```
terraform apply
```

Enter:

text

CopyEdit

```
eu-west-1
```

 EC2 instance created in **eu-west-1**

 Save the AMI ID for later comparison.

Test Case 2: Set Region via Inline CLI `-var`

bash

CopyEdit

```
terraform apply -var="aws_region=us-east-1"
```

 New instance created in **us-east-1**

 **But... WHAT HAPPENED TO THE PREVIOUS INSTANCE?**

- The **EU West** instance is still running.
- It's NOT visible to Terraform anymore.
- Why? Because:

hcl

CopyEdit

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

 This changes the *entire backend context*. The **state file now reflects only the region passed in.**

Terraform Internals: Why This Is Dangerous

- Terraform **writes the provider's region into the state metadata**.
- If you change the region, Terraform interprets it as an **entirely different environment**.
- It does not compare resources across regions.
-  It will not destroy or manage resources from the previous region anymore!

```
bash
CopyEdit
terraform state list
# shows only resources from the current region!
```

Demonstration Summary

Action	Outcome
Deploy in <code>eu-west-1</code>	Instance created
Deploy in <code>us-east-1</code> via CLI <code>-var</code>	New instance created
Check state	Shows only <code>us-east-1</code>
<code>terraform destroy</code> in <code>us-east-1</code>	Destroys only US instance
EU instance is left behind	 Not destroyed or managed!

Danger Zone: Why Region in Provider is Risky

Use Cases

- Multi-region testing
- Manually switching regions for experimentation

Misuse

- Using a **dynamic provider** for production without tracking state per region
 - Forgetting to destroy resources from earlier region
 - Confusing behavior if your team is unaware of regional switching
-

Recommendation

Always use **separate workspaces, state backends, or distinct configurations** per region when working across multiple AWS regions.

Bonus: What Happens with Bad Inputs?

Case 1: Entering a number (1) for string variable

Terraform silently converts 1 → "1"

AWS returns:

```
javascript
CopyEdit
Error: Invalid AWS Region: "1"
```

Case 2: Typing Variable as List but passing Number

```
hcl
CopyEdit
variable "aws_region" {
  type = list(string)
}
```

Run:

```
bash
```

CopyEdit
`terraform apply`

Enter:

text
CopyEdit
`1`

Terraform Error:

bash
CopyEdit
`Incorrect value type`

 Terraform **cannot convert number → list**, but it can **coerce number → string**.

Final Cleanup Steps

1. Manually delete leftover EC2 instance from `eu-west-1` (because it's unmanaged now)
2. Reset `provider.tf` to hardcoded value:

hcl
CopyEdit
`provider "aws" {
 region = "eu-west-1"
}`

Summary of Key Learnings

Topic	Detail
Input variable setup	Using <code>variables.tf</code> with type <code>string</code>
CLI behavior	Prompts for missing vars unless overridden
Ways to pass value	CLI <code>-var</code> , default, env var, varfile

Region as provider input	Changes backend context
Terraform state	Scoped per-region; not cross-region aware
Danger	Orphaned resources, inconsistent state
Safe practice	Use separate configs/workspaces per region

Full Code Snapshot

```
hcl
CopyEdit
// variables.tf
variable "aws_region" {
    type      = string
    description = "AWS region to deploy into"
}

hcl
CopyEdit
// provider.tf
terraform {
    required_version = ">= 1.7.0"

    required_providers {
        aws = {
            source  = "hashicorp/aws"
            version = ">= 5.0, < 6.0"
        }
    }
}

provider "aws" {
    region = var.aws_region
}

hcl
CopyEdit
// compute.tf
resource "aws_instance" "compute" {
    ami          = data.aws_ami.ubuntu.id
    instance_type = "t2.micro"
```

```
root_block_device {
    delete_on_termination = true
    volume_size          = 10
    volume_type          = "gp3"
}
}

data "aws_ami" "ubuntu" {
    most_recent = true

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

    filter {
        name    = "virtualization-type"
        values = ["hvm"]
    }

    owners = ["099720109477"]
}
```

GOAL

In this lecture, we move from basic input variables to advanced usage:

- Add input variables for:
 - `instance_type`
 - `volume_type`
 - `volume_size`
 - Introduce **descriptions**, **default values**, and **validation rules**
 - Prevent costly misconfigurations using **guardrails**
 - Clean up using `terraform fmt`
-

File Structure Overview

```
bash
CopyEdit
08-input-vars-locals-outputs/
├── provider.tf          # Hardcoded region
├── compute.tf            # Uses variables for EC2 config
└── variables.tf          # All input variable declarations &
                           validation
```

Step 1: Define Input Variables — `variables.tf`

```
h
CopyEdit
variable "instance_type" {
  type      = string
  default   = "t2.micro"
```

```

description = "The instance type of the managed EC2 instances."

validation {
    condition      = contains(["t2.micro", "t3.micro"],
var.instance_type)
    error_message = "Only t2.micro and t3.micro are supported EC2
instance types."
}
}

variable "volume_size" {
    type          = number
    description   = "The size (in GB) of the root block volume attached
to EC2 instances."
}

variable "volume_type" {
    type          = string
    description   = "The volume type of the root block device (e.g., gp2
or gp3)."
}

```

Explanation

type

- Ensures only appropriate data types are passed:
 - `string` for textual values like instance types and volume types
 - `number` for sizes like volume size (in GB)

default

- Prevents prompting for values if no override is provided
- Good for common, safe configurations (e.g., `t2.micro` is in free tier)

description

- Documents intent
- Shown during interactive CLI prompts — helps users know what to input

✓ validation

- `contains()` function ensures only certain values are accepted
 - Prevents accidentally spinning up expensive instances (e.g., `t3.4xlarge`)
-

Step 2: Use Input Variables in EC2 Configuration — `compute.tf`

```
hcl
CopyEdit
resource "aws_instance" "compute" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  root_block_device {
    delete_on_termination = true
    volume_size          = var.volume_size
    volume_type          = var.volume_type
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true

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

  filter {
    name    = "virtualization-type"
    values = ["hvm"]
  }
}
```

```
    owners = [ "099720109477" ] # Canonical
}
```

What Changed From Previous Lecture

Resource	Old	New using Variable
instance_type	"t2.micro"	var.instance_type
volume_size	10	var.volume_size
volume_type	"gp3"	var.volume_type

CLI Usage: Passing Variable Values

```
bash
CopyEdit
terraform apply -var="volume_size=10" -var="volume_type=gp3"
```

- `instance_type` will default to `t2.micro`, no prompt
-

Common Mistake: Passing an Expensive EC2 Instance Type

```
bash
CopyEdit
terraform apply -var="instance_type=t3.4xlarge"
-var="volume_size=10" -var="volume_type=gp3"
```

 Result:

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

 Terraform accepts it — **unless validation is defined!**

Add Validation to Prevent Expensive Instances

h

CopyEdit

```
validation {
  condition      = contains(["t2.micro", "t3.micro"],
var.instance_type)
  error_message = "Only t2.micro and t3.micro are supported EC2
instance types."
}
```

 Now `t3.4xlarge` is rejected:

text

CopyEdit

```
| Error: Invalid value for variable
|
| Only t2.micro and t3.micro are supported EC2 instance types.
```

Terraform Internal Behavior

- Validations happen **before planning**.
 - Invalid inputs are **caught early**, saving time and money.
 - If multiple validations fail, Terraform stops at the **first failure**.
 - Terraform expressions like `contains()` are **evaluated client-side**, before any API calls.
-

Bonus: Comparison of `or` vs `contains()`

Option 1 (Long):

hcl

CopyEdit

```
condition = (
    var.instance_type == "t2.micro" ||
    var.instance_type == "t3.micro"
)
```

Option 2 (Elegant):

hcl

CopyEdit

```
condition = contains(["t2.micro", "t3.micro"], var.instance_type)
```

 The `contains()` version is easier to read and scales better with more options.

Step 3: Format the Code

bash

CopyEdit

```
terraform fmt
```

 Ensures all blocks are correctly indented and styled.

Final Full Code Snapshot

variables.tf

hcl

CopyEdit

```
variable "instance_type" {
    type      = string
    default   = "t2.micro"
    description = "The instance type of the managed EC2 instances."

    validation {
        condition      = contains(["t2.micro", "t3.micro"],
var.instance_type)
        error_message = "Only t2.micro and t3.micro are supported EC2
instance types."
    }
}
```

```
}

variable "volume_size" {
  type      = number
  description = "The size (in GB) of the root block volume attached
to EC2 instances."
}

variable "volume_type" {
  type      = string
  description = "The volume type of the root block device (e.g., gp2
or gp3)."
}
```

compute.tf

```
hcl
CopyEdit
resource "aws_instance" "compute" {
  ami          = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  root_block_device {
    delete_on_termination = true
    volume_size          = var.volume_size
    volume_type           = var.volume_type
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true

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

  filter {
    name    = "virtualization-type"
    values = ["hvm"]
  }
}
```

```
    owners = ["099720109477"]  
}
```

Summary: What You Mastered

Concept	Description
<code>type</code>	Restricts allowed value types
<code>default</code>	Provides fallback when no input is given
<code>description</code>	Useful for documentation and CLI UX
<code>validation</code>	Critical for preventing misconfiguration
<code>contains()</code>	Elegant, scalable validation function
<code>terraform</code>	Keeps your code clean and consistent
<code>fmt</code>	

GOD-LEVEL DEVOPS NOTES — LECTURE 4

 Topic: Complex Terraform Variable Types (Objects, Maps, and Practical Use in EC2 Resource Configuration)

Lecture Overview

In this lecture, we moved from **primitive variables** to **complex types**—primarily focusing on:

-  Using `object` types to group related input variables
 -  Understanding the difference between `object` and `map`
 -  Practical application: Refactoring EC2 volume configuration
 -  Adding dynamic tagging via maps
 -  Usage of `merge()` to combine static and dynamic tags
 -  Caveats when using objects & maps (e.g., key presence, interpolation logic)
-

GOAL

Refactor Terraform code to:

- Use **one object** variable for EC2 volume configuration
 - Introduce a **map variable** for optional additional tags
 - Demonstrate how objects and maps affect resource definitions
 - Showcase default values, type safety, and interpolation behavior
-

INITIAL SETUP (Before Refactor)

Previously, you had **two separate primitive variables**:

hcl

```
CopyEdit
variable "ec2_volume_size" {
  type      = number
  description = "The size of the root block volume (in GB)"
}

variable "ec2_volume_type" {
  type      = string
  description = "The type of the root block volume"
}
```

In the `aws_instance` resource:

```
hcl
CopyEdit
root_block_device {
  volume_size = var.ec2_volume_size
  volume_type = var.ec2_volume_type
}
```

AFTER REFACTOR: OBJECT-BASED CONFIGURATION

1. Define an Object Variable

```
hcl
CopyEdit
variable "ec2_volume_config" {
  type = object({
    size = number
    type = string
  })
  description = "The size and type of the root block volume for EC2 instances"
  default = {
    size = 10
    type = "gp3"
  }
}
```

Why an object?

Combining `size` and `type` into one object improves cohesion, reusability, and enforces validation as one block.

2. Modify the Resource to Use `object` Fields

```
hcl
CopyEdit
resource "aws_instance" "this" {
    ami           = data.aws_ami.ubuntu.id
    instance_type = var.ec2_instance_type

    root_block_device {
        volume_size = var.ec2_volume_config.size
        volume_type = var.ec2_volume_config.type
    }

    tags = merge(
        var.additional_tags,
        {
            Name      = "ExampleInstance"
            ManagedBy = "Terraform"
        }
    )
}
```

3. Add Optional Tags with a Map

```
hcl
CopyEdit
variable "additional_tags" {
    type      = map(string)
    description = "Optional additional tags for EC2 instance"
    default    = {}
}
```

 Using `merge()` allows combining **static** and **user-defined dynamic tags**.



CLI BEHAVIOR WITH OBJECTS

When using CLI:

```
bash
CopyEdit
terraform apply \
-var='ec2_volume_config={ size = 20, type = "gp3" }'
```

Object CLI syntax must use exact structure (spaces allowed if quoted properly).

You **must** pass both `size` and `type` if no default is defined. Partial object input will error out.



Internal Terraform Mechanics

Terraform Interpolation:

```
hcl
CopyEdit
volume_size = var.ec2_volume_config.size
```

- At plan time, Terraform unpacks the object and interpolates each value.
 - If `ec2_volume_config` is undefined and no default is set → error
-



OBJECT vs MAP in Terraform

Feature	object	map
Key structure	Fixed & known (declared at type level)	Arbitrary keys (user-defined)
Usage	Structured configs (e.g., volume settings)	Optional, variable-length data (e.g., tags)
Access	<code>var.obj.key</code>	<code>var.map["key"]</code>
Validation	Enforced at declaration	No key enforcement

Common Pitfalls

Issue	Explanation
 Missing object key	Terraform will fail if <code>var.ec2_volume_config</code> is missing <code>size</code> or <code>type</code>
 Wrong types	<code>"10"</code> instead of <code>10</code> will throw an error in <code>size = number</code>
 Mismatch in key name	If you rename object fields without updating usage, interpolation fails
 Using map instead of object	Leads to runtime errors when trying to access specific keys if they're missing

Optional: Map of Objects

Terraform allows:

```
hcl
CopyEdit
variable "ec2_volume_config_map" {
  type = map(object({
    size = number
    type = string
  }))
  default = {
    config = {
      size = 10
      type = "gp3"
    }
  }
}
```

Accessing:

```
hcl
CopyEdit
var.ec2_volume_config_map["config"].size
```

 Useful when managing **multiple EC2 configs** in one variable block.

Final Code (Cleaned & Formatted)

variables.tf

hcl
CopyEdit

```
variable "ec2_volume_config" {
  type = object({
    size = number
    type = string
  })
  description = "The size and type of the root block volume for EC2 instances"
  default = {
    size = 10
    type = "gp3"
  }
}

variable "additional_tags" {
  type      = map(string)
  description = "Optional additional tags for EC2 instance"
  default    = {}
}
```

main.tf (compute block)

hcl
CopyEdit

```
resource "aws_instance" "this" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.ec2_instance_type

  root_block_device {
    volume_size = var.ec2_volume_config.size
    volume_type = var.ec2_volume_config.type
  }

  tags = merge(
    var.additional_tags,
    {

```

```
    Name      = "ExampleInstance"
    ManagedBy = "Terraform"
  }
)
}
```

Final Thoughts

-  **Use `object`** when the variable's structure is strict and fixed.
-  **Use `map`** when you want flexible, arbitrary inputs (like tags).
-  **Use `merge()`** to combine static and dynamic configurations safely.
-  **Always run `terraform fmt`** after editing.
-  **Avoid hardcoding JSON-like objects via CLI unless necessary** — prefer `.tfvars` files (covered next).

Lecture 5: TF Vars Files in Terraform

— Precise, Scalable Variable Management

Introduction: Why Use `.tfvars` Files?

Purpose:

Until now, variable values were provided via:

- Defaults in `variables.tf`
- CLI (`-var`, `-var-file`)
- Environment vars (`TF_VAR_name`)
- Interactive prompt (if no value/default)

This lecture introduces **TF vars files** — clean, reusable files that define values for input variables across environments (like `dev`, `prod`, etc.).



1. Review of Existing Variables

We are building on the following variables:

♦ `ec2_instance_type`

hcl

CopyEdit

```
variable "ec2_instance_type" {
    type      = string
    description = "Instance type to use for EC2."
    # ❌ No default, must be supplied externally (tfvars, CLI, etc.)
    validation {
        condition = contains(["t2.micro", "t3.micro", "t3.large"],
var.ec2_instance_type)
```

```
        error_message = "Allowed values are t2.micro, t3.micro, or  
t3.large."  
    }  
}
```

- ◆ **ec2_volume_config (an object)**

```
hcl  
CopyEdit  
variable "ec2_volume_config" {  
    type = object({  
        size = number  
        type = string  
    })  
    default = {  
        size = 10  
        type = "gp3"  
    }  
    description = "Configuration for root EBS volume (size in GiB and  
type)."  
}
```

- ◆ **additional_tags (a map)**

```
hcl  
CopyEdit  
variable "additional_tags" {  
    type      = map(string)  
    default   = {}  
    description = "Optional additional tags for EC2 instances."  
}
```

Why use a map instead of an object for tags?

- **Map** allows arbitrary keys.
- **Object** requires a fixed schema.
- Tags vary per environment, so `map(string)` is the right fit.

2. Creating the `terraform.tfvars` File

File: `terraform.tfvars`

hcl
CopyEdit
`ec2_instance_type = "t2.micro"`

`ec2_volume_config = {
 size = 10
 type = "gp2"
}`

`additional_tags = {
 source = "terraform.tfvars"
}`

Internal Terraform Behavior

- Terraform **automatically detects** and loads a file named **exactly** `terraform.tfvars` or ending with `.auto.tfvars`.
 - All variable values in that file override:
 - Defaults in `variables.tf`
 - Environment variables
 - Does *not* override values passed via CLI (which have highest precedence).
-

Run: `terraform plan`

bash
CopyEdit
`terraform plan`

CLI Output Highlights:

```
text
CopyEdit
+ instance_type = "t2.micro"
+ root_block_device {
    volume_size = 10
    volume_type = "gp2"
}
+ tags = {
    "managed_by" = "Terraform"
    "source"      = "terraform.tfvars"
}
```

Interpretation:

- All variable values are sourced from `terraform.tfvars`.
- Overwrites the default of `gp3` (now using `gp2`) → confirmed by root block output.
- Tags include:
 - Static tag (`managed_by = "Terraform"`)
 - Dynamic tag from `var.additional_tags` map (`source = terraform.tfvars`)

3. Changing the File Name → Behavior Changes

Rename File: `dev.terraform.tfvars`

```
bash
CopyEdit
mv terraform.tfvars dev.terraform.tfvars
```

Run: `terraform plan`

```
bash
CopyEdit
terraform plan
```

CLI Output:

```
text
CopyEdit
var.ec2_instance_type
Enter a value:
```

Explanation:

Terraform no longer finds `terraform.tfvars` → prompts for missing required variables (`ec2_instance_type` has no default!).

Fix: Use -var-file Flag

```
bash
CopyEdit
terraform plan -var-file="dev.terraform.tfvars"
```

 Terraform loads the file and uses values as before.

4. Adding a prod.terraform.tfvars File

File: prod.terraform.tfvars

```
hcl
CopyEdit
ec2_instance_type = "t3.large"

ec2_volume_config = {
  size = 50
  type = "gp3"
```

```
}

additional_tags = {
  source = "prod.terraform.tfvars"
}
```

 **Run: terraform plan**
-var-file="prod.terraform.tfvars"

 **Output:**

- Instance type: `t3.large`
- Volume: `50 GiB`, type `gp3`
- Tags:
 - `managed_by = "Terraform"`
 - `source = "prod.terraform.tfvars"`

Terraform Precedence Order (Variable Sources)

Priority	Source	Overwrites
①	CLI: <code>-var</code> or <code>-var-file</code>	All others
②	Environment var: <code>TF_VAR_name</code>	Below CLI
③	<code>.auto.tfvars</code> or <code>terraform.tfvars</code>	Below env
④	Default in <code>variables.tf</code>	Lowest

 Terraform uses the **highest precedence** value when multiple sources define the same variable.

Internal Behavior and Terraform State Impact

- Terraform stores all resolved variable values in `.terraform/terraform.tfstate`.
- Even `sensitive = true` values are stored in plain text in the state file.
- Each `.tfvars` file changes the rendered plan (which affects the state if `apply` is executed).
- Switching `.tfvars` files without changing environment leads to drift/conflict unless isolated by:



Solution: Terraform Workspaces (coming up in future lectures)

5. Structuring Environments: When to Use Multiple `.tfvars`

Use multiple `.tfvars` files when:

- You manage multiple environments from the same codebase (e.g., `dev`, `prod`, `staging`)
- Each environment needs different:
 - EC2 instance types
 - EBS volumes
 - Tags
 - Region, VPC IDs, etc.

⚠ Common Mistakes

Mistake	Why it fails
✗ Renaming <code>terraform.tfvars</code> to custom name without <code>-var-file</code>	Terraform won't pick it up automatically
✗ Providing wrong types (e.g., string instead of object)	Type validation will fail
✗ Using map when object is required (or vice versa)	May result in missing key errors
✗ Forgetting to validate <code>.tfvars</code> structure	Leads to runtime errors during <code>plan</code>



Tips for Professional Projects

- Keep environment-specific `.tfvars` files in `/environments/dev/`, `/environments/prod/`
- Use `terraform plan -var-file=...` in CI/CD pipelines
- Name `.tfvars` consistently (`dev.tfvars`, `prod.tfvars`)
- Version control all `.tfvars` files — **except** those with secrets (use `.gitignore` or Vault)



Final Codebase Snapshot

◆ `variables.tf`

(unchanged from previous lecture, but you could annotate this if needed)

◆ `dev.terraform.tfvars`

```
hcl
CopyEdit
ec2_instance_type = "t2.micro"
```

```
ec2_volume_config = {
    size = 10
    type = "gp2"
}

additional_tags = {
    source = "dev.terraform.tfvars"
}
```

◆ **prod.terraform.tfvars**

hcl
CopyEdit
`ec2_instance_type = "t3.large"`

```
ec2_volume_config = {
    size = 50
    type = "gp3"
}

additional_tags = {
    source = "prod.terraform.tfvars"
}
```

Summary (Not a summary of the lecture — just of key behaviors)

- `.tfvars` files are the **recommended** method for variable management.
- `terraform.tfvars` and `*.auto.tfvars` are loaded **automatically**.
- Custom-named files must be passed explicitly using `-var-file`.
- Use **maps** for flexible structures, **objects** for strict schemas.
- Future fix for env-isolation = **Terraform Workspaces**.

Lecture 6 – Terraform Auto `.tfvars` Files & Override Precedence

Goal of Lecture

Learn how to:

- Work with `terraform.tfvars`
 - Use `.auto.tfvars` files to override specific variables
 - Understand **variable override precedence**
 - Handle **multiple auto files** and **avoid confusion**
 - Use **safe instance types** to avoid billing charges
-

Step-by-Step Summary

Step 1: Rename and Clean Up Existing Files

1. Rename:
 - Rename `production.tfvars` → `terraform.tfvars`
 - Purpose: Stick to **default naming conventions**
 2. Delete:
 - Remove unnecessary custom `.tfvars` files for simplicity
 - Keep only `terraform.tfvars` to demonstrate automatic behavior
-

Step 2: How Terraform Uses `terraform.tfvars`

Run:

```
bash
CopyEdit
terraform plan
```

- - Result:
 - Terraform **automatically picks up `terraform.tfvars`** if no `-var-file` is explicitly passed.
 - This is part of **Terraform's default behavior**.
-



Step 3: Create and Use `.auto.tfvars` File

Create a new file:

```
plaintext
CopyEdit
prod-override.auto.tfvars
```

1.

Add the following inside:

```
hcl
CopyEdit
ec2_instance_type = "t3.micro"
```

2.

Run:

```
bash
CopyEdit
terraform plan
```

3.

4. Terraform Behavior:

- Terraform automatically detects any file ending with `.auto.tfvars`

- Overrides the matching variable (`ec2_instance_type`) from `terraform.tfvars`
-

Precedence Behavior

- When the **same variable** exists in multiple `.tfvars` files:
 - `.auto.tfvars` overrides `terraform.tfvars`
 - Only the **matching variable** is overridden, not the entire file

 Overrides happen at the variable level, NOT the file level

 Example:

- If `prod-override.auto.tfvars` only contains `ec2_instance_type`, then:
 - `ec2_instance_type` comes from `prod-override.auto.tfvars`
 - All other values still come from `terraform.tfvars`

Step 4: Testing Multiple `.auto.tfvars` Files

Create a second override file:

```
plaintext
CopyEdit
dev-override.auto.tfvars
```

1.

Add inside:

```
hcl
CopyEdit
additional_tags = {
  Environment = "Development"
}
```

2.

The existing `prod-override.auto.tfvars` contains:

```
hcl
CopyEdit
additional_tags = {
  Environment = "Production"
}
```

3.

Run:

```
bash
CopyEdit
terraform plan
```

4.



Lexical Order Precedence

Terraform processes `.auto.tfvars` files in **lexical order (alphabetical order by filename)**:

File Name	Lexical Order
<code>dev-override.auto.tf</code>	1st
<code>vars</code>	
<code>prod-override.auto.t</code>	2nd
<code>tfvars</code>	
•	
•	! Last override wins
•	Hence, the final value of <code>additional_tags</code> will come from <code>prod-override.auto.tfvars</code>



Warning: Dangers of Multiple `.auto.tfvars`

- Using **multiple `.auto.tfvars`** files can be **very confusing**
- It's hard to determine the final values being passed to Terraform
- Not beginner-friendly or team-friendly

Best Practice:

 **Stick to one `.auto.tfvars` file if possible**

Or explicitly pass `.tfvars` files using `-var-file` to avoid ambiguity

Step 5: Cleanup

Delete `dev-override.auto.tfvars` to reduce confusion:

```
bash
CopyEdit
rm dev-override.auto.tfvars
```

- Rename:
 - Rename `prod-override.auto.tfvars` → `prod.auto.tfvars`

Billing Advice: Use Free Tier Instance Types

Free Tier EC2 types:

Instance Type	Free Tier	Region Support
<code>t2.micro</code>	 Yes	Common
<code>t3.micro</code>	 Sometimes	In regions without <code>t2.micro</code>

 Check your AWS region:

Deploying `t3.micro` in a region that supports `t2.micro` may incur charges.

Safe Approach:

```
hcl
CopyEdit
ec2_instance_type = "t2.micro"
```

- Use this in `.auto.tfvars` to avoid unexpected billing
 - Confirm availability in your region before applying
-

Summary of Key Learnings

Topic	Notes
<code>terraform.tfvars</code>	Auto-loaded, used for default variable values
<code>.auto.tfvars</code>	Also auto-loaded, overrides same variables in <code>terraform.tfvars</code>
Precedence	Lexical order among <code>.auto.tfvars</code> files determines which wins
Variable-level override	Only the specific variable is overridden, not the entire file
Best Practice	Avoid multiple <code>.auto.tfvars</code> unless absolutely necessary
Billing Tip	Use <code>t2.micro</code> to stay within AWS Free Tier

Pro Tip

If you must work with multiple `.tfvars`:

```
bash
CopyEdit
terraform plan -var-file="prod.tfvars"
```

- This approach is **explicit, predictable, and team-friendly**
- Better than relying on `.auto.tfvars` precedence in complex setups



Lecture 7: Terraform Input Variable Precedence (GOD-LEVEL NOTES)



Overview

This lecture explores **how Terraform determines which values to use** when multiple sources provide input for the same variable. You'll learn:

- How to override Terraform input variables from different sources
 - The **exact order of precedence**
 - Internal behavior when processing each method
 - Best practices and safety guidance to avoid cost issues
 - CLI walkthroughs with exact output and analysis
-



Variable Validation Change (Temporarily Adjusted)

Before we test precedence, we modify the validation rule for the `ec2_instance_type` input variable.



Old Validation:

```
hcl
CopyEdit
validation {
  condition      = contains(["t2.micro", "t3.micro"],
var.ec2_instance_type)
  error_message = "Only t2.micro or t3.micro is allowed."
}
```



Temporary Validation: `startswith()`

```
hcl
CopyEdit
validation {
  condition      = startswith(var.ec2_instance_type, "t3.")
```

```
    error_message = "Only supports T3 family."  
}
```

 This allows us to test multiple values (e.g. `t3.micro`, `t3.small`, `t3.large`, etc.) while keeping validation logic flexible.

1. Environment Variable Input

Command:

bash
CopyEdit
`export TF_VAR_ec2_instance_type=t3.micro`

 `TF_VAR_` is the **Terraform environment variable convention**. Terraform automatically maps this to a declared variable of the same name (`ec2_instance_type`).

Internal Behavior:

- Environment variables are **the lowest in the precedence hierarchy**
 - Accessible only in the current shell session
 - Safe for **sensitive values** that you don't want to store in versioned files
-

2. `terraform.tfvars` File

File: `terraform.tfvars`

hcl
CopyEdit
`ec2_instance_type = "t3.small"`

Internal Logic:

- Terraform **automatically loads** `terraform.tfvars` (no need to specify it)

- Overrides environment variables

Command:

```
bash
CopyEdit
terraform plan | grep instance_type
```

Output:

```
arduino
CopyEdit
+ instance_type = "t3.small"
```

 Even though `TF_VAR_ec2_instance_type` was exported, the value in `terraform.tfvars` wins.

3. `prod.auto.tfvars` File

File: `prod.auto.tfvars`

```
h
CopyEdit
ec2_instance_type = "t3.large"
```

Internal Logic:

- Any file with `.auto.tfvars` extension is loaded automatically
- Lexical order determines precedence when multiple auto files exist
- Overrides both:
 - `terraform.tfvars`
 - Environment variables

Command:

```
bash
CopyEdit
```

```
terraform plan | grep instance_type
```

Output:

```
arduino
CopyEdit
+ instance_type = "t3.large"
```

4. CLI - **-var** Flag

Command:

```
bash
CopyEdit
terraform plan -var="ec2_instance_type=t3.xlarge" | grep
instance_type
```

Internal Behavior:

- Command-line flags **take precedence over every other input source**
- Last one passed **wins if multiple -var flags are used**

Output:

```
arduino
CopyEdit
+ instance_type = "t3.xlarge"
```

 Internal Terraform engine evaluates CLI values **last** in the graph build phase, giving them absolute priority.

5. CLI - **-var-file** Flag

File: **override.tfvars**

```
hcl
CopyEdit
ec2_instance_type = "t3.2xlarge"
```

Command:

bash

CopyEdit

```
terraform plan -var-file=override.tfvars | grep instance_type
```

Output:

arduino

CopyEdit

```
+ instance_type = "t3.2xlarge"
```

 Since this is passed **via CLI**, it also **overrides all previous sources**, just like `-var`.

Full Precedence Order (Highest to Lowest):

Priority	Method	Example
1	CLI <code>-var</code>	<code>-var="key=value"</code>
2	CLI <code>-var-file</code>	<code>-var-file=override.tfvars</code>
3	<code>.auto.tfvars</code> Files	<code>prod.auto.tfvars</code>
4	<code>terraform.tfvars</code>	File auto-loaded by Terraform
5	Environment Variables	<code>export TF_VAR_key=value</code>
6	Default values in variable block	<code>default = "value"</code>

 **Lexical order** applies to multiple `.auto.tfvars` files. For example, `dev.auto.tfvars` overrides `prod.auto.tfvars` because **d < p**.

Gotchas and Edge Cases

Sensitive Info in `.tfvars` Files

- **Never** store secrets in `.tfvars`, `.auto.tfvars`, or `terraform.tfvars`.
- Use `TF_VAR_` from environment or secrets manager pipelines (like AWS Secrets Manager + CI/CD).

Overlapping `.auto.tfvars` Files

plaintext
CopyEdit
`dev.auto.tfvars`
`prod.auto.tfvars`

- Lexical processing (`dev` before `prod`)
 - The **last file read wins** for a given variable key
 - **Avoid** this pattern: it creates unpredictable and hard-to-debug configurations
-

Cleanup (Best Practice)

Before wrapping up:

- Deleted `prod.auto.tfvars` to avoid surprise auto-loaded values
- Reset `terraform.tfvars` to use safe free-tier value:

hcl
CopyEdit
`ec2_instance_type = "t2.micro" # Free tier`

- Left `override.tfvars` for **manual CLI testing only**:

bash
CopyEdit
`terraform plan -var-file=override.tfvars`

Final Variable Configuration (Safe State)

File	Contains	Picked Up Automatically?
terraform.tfvars	t2.micro ars	
override.tfvars	t3.micro or other	 (only via <code>-var-file</code>)
prod.auto.tfvars	 (Deleted)	N/A

Summary: Use Cases for Each Input Method

Use Case	Recommended Input Method
Temporary override during dev	<code>-var</code> CLI flag
Parameter file in CI/CD	<code>-var-file</code>
Environment-specific values	<code>terraform.tfvars</code> (dev/test)
Secret or sensitive values	<code>TF_VAR_x</code> env vars
Avoiding confusion	 Stick to one source per env



Final Tips

-  **Explicit > Implicit:** Prefer using `-var-file` or `-var` instead of `.auto.tfvars`
-  Test plans with `grep` to inspect values before applying
-  Use free-tier instance types (`t2.micro`, `t3.micro`) to avoid charges



Lecture 8: Mastering `locals` in Terraform — Tag Refactoring & S3 Resource Creation



Objective

Refactor an existing Terraform configuration using `locals` for DRY (Don't Repeat Yourself) best practices. Use locals for:

- 📦 Tag reusability
 - 🏙 Resource naming (including interpolation)
 - 💧 Deploying a new S3 bucket using local-based logic
 - ✎ Generating a unique bucket suffix via `random_id`
-



What Are `locals` in Terraform?



Definition

`locals` define **internal-only values** in a Terraform configuration.

```
hcl
CopyEdit
locals {
  key = value
}
```



Key Properties

- Locals **cannot be set from the outside** (e.g. via CLI or tfvars)
- Ideal for **intermediate values**, computed defaults, and DRY logic
- Can reference:

- Variables: `var.xyz`
 - Other locals: `local.abc`
 - Functions/expressions: `join()`, `lookup()`, etc.
 - Used in outputs, resource names, tags, etc.
-

Step-by-Step Breakdown

1 Define Basic Locals

 File: `compute.tf`

```
hcl
CopyEdit
locals {
  project_owner = "Terraform course"
  cost_center   = "1234"
  managed_by    = "Terraform"
}
```

These are simple, **primitive values (string)** used to tag resources.

2 Define Composite Local for Tags

```
hcl
CopyEdit
locals {
  common_tags = {
    project_owner = local.project_owner
    cost_center   = local.cost_center
    managed_by    = local.managed_by
  }
}
```

 You can reference other `local.X` inside another `locals` block. They're all **merged during plan time**.

3 Refactor EC2 Instance Tags

 File: `compute.tf`

```
hcl
CopyEdit
tags = merge(local.common_tags, var.additional_tags)
```

Behavior

- Uses `merge()` to combine:
 - `local.common_tags` (internal)
 - `var.additional_tags` (user-provided)
- Order matters: **user overrides common if keys overlap**

CLI Check

```
bash
CopyEdit
terraform plan | grep tags
```

4 Define a `project` Local for Naming

```
hcl
CopyEdit
locals {
  project = "08-input-vars-locals-outputs"
}
```

Used for:

- Naming S3 buckets
- Tags (`project`)

5 Add New Resource: S3 Bucket

 File: `s3.tf`

```
hcl
CopyEdit
resource "aws_s3_bucket" "project_bucket" {
  bucket = "${local.project}"
  tags   = merge(local.common_tags, var.additional_tags)
}
```

Terraform Plan Output (partial)

```
hcl
CopyEdit
+ bucket = "08-input-vars-locals-outputs"
+ tags = {
    "project_owner" = "Terraform course"
    ...
}
```

Internal Behavior

- Terraform **resolves all locals** before execution
- `merge()` works only on maps
- Bucket name becomes static unless we add randomness

Adding Randomness to Bucket Name

6 Add `random` Provider

 File: `providers.tf` or `main.tf`

```
hcl
CopyEdit
terraform {
  required_providers {
```

```
    random = {
      source  = "hashicorp/random"
      version = ">= 3.0.0"
    }
  }
}
```

✓ Re-initialize Terraform

bash
CopyEdit
`terraform init`

7 Add `random_id` Resource

File: `s3.tf`

hcl
CopyEdit
`resource "random_id" "project_bucket_suffix" {
 byte_length = 4
}`

⚠ `byte_length = 4` generates an 8-character hex string (`2n` length)

8 Combine Local + Random ID in Bucket Name

hcl
CopyEdit
`resource "aws_s3_bucket" "project_bucket" {
 bucket = "${local.project}-${random_id.project_bucket_suffix.hex}"
 tags = merge(local.common_tags, var.additional_tags)
}`

✓ CLI Plan Output

h
CopyEdit

```
+ bucket = "08-input-vars-locales-outputs-a3f9e2d1" # Value known  
after apply
```

 Bucket name cannot be known during `plan` — due to `random_id` resource
Terraform sets it to `known after apply`

Terraform Apply

```
bash  
CopyEdit  
terraform apply  
# Output:  
# Plan: 3 to add, 0 to change, 0 to destroy  
# aws_instance.ec2 (recreated with new tags)  
# aws_s3_bucket.project_bucket  
# random_id.project_bucket_suffix
```

Optional: Organize Locals into a Separate File

 Create: `shared-locals.tf`

Move all `locals` blocks here:

```
hcl  
CopyEdit  
locals {  
    project      = "08-input-vars-locales-outputs"  
    project_owner = "Terraform course"  
    cost_center   = "1234"  
    managed_by    = "Terraform"  
    common_tags = {  
        project_owner = local.project_owner  
        cost_center   = local.cost_center  
        managed_by    = local.managed_by  
    }  
}
```

-  No need for `import` between `.tf` files. Terraform automatically merges them into a single configuration tree.
-



Advanced Use Case Example (for later)

You can define a local that **derives from variables**:

```
hcl
CopyEdit
locals {
  ec2_instance_type = var.ec2_instance_type
}
```

 Good for:

- Preprocessing
 - Filtering
 - Using inside expressions/functions later
-



Common Mistakes

Mistake	Consequence
 Referencing local before it exists	Terraform will error
 Naming bucket without randomness	Might cause <code>BucketAlreadyExists</code> AWS error
 Defining duplicate <code>locals</code> keys in multiple blocks	Will silently overwrite in order of appearance
 Forgetting to <code>terraform init</code> after adding provider	CLI will crash or complain
 Overlapping keys in <code>merge()</code>	Later keys overwrite earlier ones silently



Cleanup

Run:

```
bash
CopyEdit
terraform destroy
```

Output:

```
plaintext
CopyEdit
Plan: 3 to destroy
- random_id.project_bucket_suffix
- aws_instance.ec2
- aws_s3_bucket.project_bucket
```

Key Takeaways

Feature	Use
<code>locals</code>	Internal-only values (not user-settable)
<code>merge()</code>	Combine maps (e.g. tags)
<code>random_id</code>	Ensure unique names for global resources
<code>project</code> local	Consistent naming strategy
Shared locals file	Optional structure for better organization

Lecture 9: Deep Dive into Terraform Outputs

Objective

- Create an output that exposes the **S3 bucket name** created via Terraform.
 - Understand how to use and consume Terraform outputs both within Terraform and externally.
 - Learn how to **access outputs** using CLI commands.
 - Explore **Terraform state interaction** and how outputs relate to **remote backends** (e.g., S3).
-

Step-by-Step: Working with Terraform Outputs

1. Initial Setup

The goal is to focus on the S3 bucket resource only. We **remove (or comment out)** the EC2 instance resource to simplify the working example.

```
hcl
CopyEdit
# Commenting out EC2 to focus on S3
# resource "aws_instance" "my_ec2_instance" {
#   ...
# }
```

 Purpose: Reducing resource noise to isolate output behavior.



2. Terraform Plan & Apply Without Outputs

Run:

```
bash
CopyEdit
terraform plan
terraform apply -auto-approve
```



Output shows:

- Bucket name is created with a **random suffix**
- However, **this value is not accessible outside** of Terraform (e.g., CI/CD scripts, humans)

 *We see values in the CLI after apply, but they are not reusable unless explicitly exposed.*



3. Create Output in `outputs.tf`

 Create a new file `outputs.tf` (recommended convention for better organization)

```
hcl
CopyEdit
output "s3_bucket_name" {
  value      = aws_s3_bucket.project_bucket.bucket
  description = "The name of the S3 bucket"
}
```

 `aws_s3_bucket.project_bucket.bucket` is an **attribute** of the `aws_s3_bucket` resource

You can hover in editors like VSCode or run `terraform console` to inspect all available attributes.



4. Re-run Terraform Apply

```
bash
CopyEdit
```

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

⚠ You will now see a new **Outputs:** section:

makefile

CopyEdit

Outputs:

```
s3_bucket_name = "08-input-vars-locals-outputs-r45g"
```

💡 This means Terraform now exposes the bucket name via output.

🧪 5. Accessing Output via CLI

a. Standard Output

bash

CopyEdit

```
terraform output s3_bucket_name
```

Result:

arduino

CopyEdit

```
"08-input-vars-locals-outputs-r45g"
```

◆ String is wrapped in quotes — could be problematic in shell scripts (e.g., curl)

b. Raw Output

bash

CopyEdit

```
terraform output -raw s3_bucket_name
```

Result:

python

CopyEdit

08-input-vars-locals-outputs-r45g

 Removes double quotes — safe for CLI interpolation (e.g., curl commands)

c. JSON Output

bash
CopyEdit
`terraform output -json`

Returns:

```
json
CopyEdit
{
  "s3_bucket_name": {
    "sensitive": false,
    "type": "string",
    "value": "08-input-vars-locals-outputs-r45g"
  }
}
```

 Useful for machine parsing or piping into `jq`

d. Output via `jq`

Assuming `jq` is installed:

bash
CopyEdit
`terraform output -json | jq -r '.s3_bucket_name.value'`

Returns:

```
python
CopyEdit
08-input-vars-locals-outputs-r45g
```

 Great for CI/CD pipelines to fetch values dynamically

6. Understanding Output Metadata

From `-json` output:

```
json
CopyEdit
{
  "sensitive": false,
  "type": "string",
  "value": "..."
}
```

- `sensitive`: If `true`, value will be redacted in logs/CLI
 - `type`: Can be `string`, `map`, `object`, `list`, etc.
 - `value`: Actual content exposed
-

7. Terraform State Interaction

Outputs are stored in the Terraform state.

By default:

```
bash
CopyEdit
.terraform/terraform.tfstate
```

If you use a **remote backend** (e.g., **S3**), outputs are stored **remotely**, making them accessible across:

- Machines
- CI/CD pipelines
- Team members

 Example (S3 backend configured in provider block):

```
hcl
CopyEdit
terraform {
  backend "s3" {
    bucket = "tf-state-bucket"
    key    = "dev/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Now `terraform output` reads from remote state — useful in:

- Jenkins, GitHub Actions
 - Environment promotion (dev → stage → prod)
-



8. Using Outputs Across Modules (Sneak Peek)

Outputs from one module can be **input to another**.

Example:

```
hcl
CopyEdit
module "network" {
  source = "./network"
}

module "web" {
  source        = "./web"
  vpc_id       = module.network.vpc_id  # <-- output of
'network'
  s3_bucket_name = module.network.s3_bucket_name
}
```



This modular approach builds complex infra from reusable units.



9. Common Mistakes & Edge Cases

Mistake

Output value references a wrong name

Plan/apply fails with `null` or `unknown attribute`

Output is marked sensitive, but you expect it in logs

CLI redacts output — won't show without additional logging

Using `terraform output` before apply

Returns: `The output value is not known`

Using `-raw` with non-string output

Will cause errors — `-raw` only works with string outputs

Behavior

Plan/apply fails with `null` or `unknown attribute`

Output is marked sensitive, but you expect it in logs

CLI redacts output — won't show without additional logging

Using `terraform output` before apply

Returns: `The output value is not known`

Using `-raw` with non-string output

Will cause errors — `-raw` only works with string outputs



Internal Terraform Behavior

- **Outputs are evaluated *after apply***, once all dependencies (like S3 bucket name) are created.
 - **Stored in state file (`terraform.tfstate` or remote backend)**
 - **No reordering necessary across files** (e.g., `outputs.tf`, `main.tf`) — Terraform scans the entire config directory.
-



Final Code

`outputs.tf`

```
hcl
CopyEdit
output "s3_bucket_name" {
  value      = aws_s3_bucket.project_bucket.bucket
  description = "The name of the S3 bucket"
}
```



10. Cleanup

Run:

```
bash
CopyEdit
terraform destroy -auto-approve
```

Destroy icon: A small orange icon of a paintbrush with a checkmark.

Destroy: Destroys:

- S3 bucket
 - Random suffix
 - State updates accordingly
-

Summary (NO TL;DR — Full Details Below)

Why use outputs?

- Make resource attributes accessible *after apply*
- Enable downstream consumption (scripts, pipelines, other modules)

How are outputs used?

- Via CLI: `terraform output [name]`
- Via automation: `terraform output -json | jq`
- Across modules via references: `module.<name>.<output>`



Lecture 10: Terraform – Sensitive Values Deep Dive



Goal:

Understand how `sensitive = true` works for **outputs** and **input variables** in Terraform — what it does **and** what it **does not** do.



What Sensitive Actually Means

- ! Marking a variable/output as `sensitive` *only redacts it from CLI/log display.*
 - ✓ It does **not encrypt or hide** the value in the Terraform state.
 - ⚠ Don't treat it as a security mechanism.
-



Sensitive Outputs



Syntax:

```
h
CopyEdit
output "s3_bucket_name" {
  value      = aws_s3_bucket.project_bucket.bucket
  sensitive = true
}
```



Effects:

- Terraform **masks the value** in CLI output/logs (`value = <sensitive>`).
- But you can still:
 - Run `terraform output s3_bucket_name` → prints the value.
 - Run `terraform output -json` → shows the sensitive flag and the actual value.
 - Check the `.tfstate` file → value is stored **in plaintext**.

 Conclusion: `sensitive = true` is purely **cosmetic/logical**, not a security boundary.

Sensitive Variables

Syntax:

```
hcl
CopyEdit
variable "my_sensitive_value" {
    type      = string
    sensitive = true
}
```

- No `default` → Terraform prompts user (CLI input **is hidden** from display).

Can also pass via:

```
bash
CopyEdit
terraform apply -var="my_sensitive_value=supersecret"
```

-
-

Sensitive Variable in Outputs

If you try to output a sensitive variable without marking the output as sensitive:

```
h
CopyEdit
output "my_sensitive_output" {
    value = var.my_sensitive_value
}
```

- Terraform will throw an error:

"Output refers to sensitive data but is not marked as sensitive."

Fix:

```
hcl
```

```
CopyEdit
output "my_sensitive_output" {
  value      = var.my_sensitive_value
  sensitive = true
}
```



Sensitive Data in Tags

You **can** pass sensitive variables into tags or other computed blocks:

```
hcl
CopyEdit
locals {
  common_tags = {
    "sensitive_tag" = var.my_sensitive_value
  }
}
```

Terraform attempts to redact sensitive values in the **plan output**, but...

Sometimes it leaks the value in diff previews or plan logs — **this is likely a bug.**

Don't rely on masking to protect secrets in logs.



Sensitive Data in the `.tfstate` File

- Stored in **plain text**, even if marked `sensitive = true`.
- Example from `.tfstate`:

```
json
CopyEdit
"outputs": {
  "my_sensitive_output": {
    "value": "supersecret",
    "sensitive": true
  }
}
```

Best Practice:

- Always encrypt and restrict access to state files.
 - Use remote backends like S3 + KMS encryption + access control.
-

Cleanup: Terraform Destroy

When destroying resources that use sensitive inputs:

```
bash
CopyEdit
terraform destroy
```

Terraform will prompt for the sensitive value again (input is masked as you type).

Alternatively:

```
bash
CopyEdit
terraform destroy -var="my_sensitive_value=supersecret"
```

Summary Table

Type	Mark as Sensitive	Redacts in CLI?	Stored Securely?	Usable via CLI/API?
output			 (plaintext)	 (raw/json)
variable		 (input masked)	 (plaintext)	
In tags/blocks		 Sometimes		

Pro Tips

- Always treat Terraform state as **sensitive data**.

- **Never assume `sensitive = true` provides security** — it's a UI/UX feature, not an encryption layer.
 - Consider using secrets managers (e.g., AWS Secrets Manager) and securely pass secrets into Terraform via environment variables or CI/CD.
-

Actions After This Lecture

-  Store `.tfstate` remotely with encryption (e.g., S3 + KMS).
-  Mark outputs/vars as sensitive where appropriate.
-  Avoid printing sensitive data anywhere in locals or tags if not absolutely needed.
-  Verify what gets redacted in your plan/output logs.

Expressions and functions

TERRAFORM EXPRESSIONS & OPERATORS — LECTURE 1 — GOD-LEVEL DETAILED NOTES

Lecture Start: Instructor's Goal

“Let's start discussing expressions in this lecture. I'm going to start at the IDE because this is a highly practical section.”

Explanation:

The instructor is diving directly into **Terraform Expressions** — key building blocks used in everything from defining variables, outputs, and resource properties to controlling logic (e.g., count conditions, map manipulation).

 Expressions are values computed from constants, variables, functions, and operators.

Folder & File Setup

“Let's create a new folder here I'm going to call this zero nine expressions.”

```
bash
CopyEdit
mkdir 09-expressions && cd 09-expressions
```

“Let's create a new file here. I'm just going to call this `locals.tf`.”

```
bash
CopyEdit
touch locals.tf
```

 `locals.tf` → Used to define **local variables** via the `locals` block, perfect for testing logic without deploying actual resources.

“We also should provide a terraform block. So `provider.tf`.”

```
bash
```

CopyEdit

```
touch provider.tf
```

📁 **provider.tf** → Contains the Terraform `terraform {}` block and provider configuration.



terraform block (in `provider.tf`)

hcl

CopyEdit

```
terraform {  
    required_version = ">= 1.7, < 2.0"  
}
```

🔍 **Explanation:**

- Pins Terraform version to 1.7 or higher but strictly less than 2.0.
- Helps avoid future breaking changes in syntax or logic.

✓ **Why?** Terraform major versions often introduce changes incompatible with previous versions.



Locals for Operators — `locals.tf`

▶ **Creating `locals.tf`**

“The first thing I want to discuss with you is that Terraform offers several operators, such as mathematical operators...”

✓ **Math Operators**

h

CopyEdit

```
locals {  
    math = 2 * 2  
}
```

Explanation:

- `2 * 2` performs basic multiplication and stores the value (`4`) in the local variable `math`.
- Math operators in Terraform include: `+`, `-`, `*`, `/`, `%`, unary `-`.

“You can also use division operator...”

```
hcl
CopyEdit
math_division = 10 / 2 # 5
```

“The plus of course the minus here and also the modulo operator.”

```
hcl
CopyEdit
math_add      = 2 + 3      # 5
math_subtract = 10 - 4      # 6
math_modulo   = 9 % 4      # 1
```

“If you add a minus here, this is going to multiply by -1.”

```
hcl
CopyEdit
math_negate = -7    # -7
```

Concepts:

- All math operators return a `number`.
- No data type declaration needed — HCL infers it automatically.

Equality Operators

“We have equality operations — equal sign or different signs.”

```
hcl
CopyEdit
equality_true  = 2 == 2      # true
equality_false = 2 != 2      # false
```

 `==` checks **equality**, `!=` checks **inequality**. Returns a **boolean**.

 “Math operations return number. Equality returns boolean.”

Comparison Operators

“Then we have comparison operators — greater than, less than...”

```
hcl
CopyEdit
comparison_gt  = 2 > 1    # true
comparison_gte = 2 >= 1   # true
comparison_lt   = 2 < 1    # false
comparison_lte  = 2 <= 2   # true
```

 Used to compare numeric values — useful for conditions, thresholds, etc.

Logical Operators

“Last but not least we have logical operators — or, and, not.”

```
h
CopyEdit
logical_not = !true          # false
logical_or  = true || false   # true
logical_and = true && false   # false
```

Boolean logic rules:

- `!` (NOT): inverts the truthiness
- `||` (OR): true if **at least one** operand is true
- `&&` (AND): true only if **both** operands are true

“If you have two conditions, use `&&`. If either works, use `||`.”

Output Block

"If you want to visualize the result, we can just add an output."

```
hcl
CopyEdit
output "operators" {
  value = {
    math      = local.math
    equality  = local.equality_false
    comparison = local.comparison_lt
    logical    = local.logical_or
  }
}
```

 Outputs help visualize **evaluated expression results** at runtime.

 Typo noted in lecture:

```
hcl
CopyEdit
# incorrect
output put
```

Should be:

```
h
CopyEdit
output "operators" { ... }
```

Terraform Formatting

"Let's run terraform fmt to align the code."

```
bash
CopyEdit
terraform fmt
```

 Aligns `=` and cleans code layout for HCL.

Terraform Init + Plan

Initialize Terraform

bash
CopyEdit
`terraform init`

Expected Output:

nginx
CopyEdit
`Terraform has been successfully initialized!`

 Sets up Terraform plugins, providers, and modules (if any).

Run Terraform Plan

bash
CopyEdit
`terraform plan`

 Output:

hcl
CopyEdit
Outputs:

```
operators = {
    "comparison" = false
    "equality" = false
    "logical" = true
    "math" = 4
}
```

 Detailed breakdown:

Key	Expression	Value	Reason
	n		
math	<code>2 * 2</code>	4	Simple multiplication

```
equality 2 != 2      false  2 equals 2
comparis 2 < 1      false  False comparison
on
logical   `true          false`
```

Instructor Notes & Ending Comments

“That’s as far as we will go with primitive operators.”

“There are also functions like `log`, `pow` for more advanced math...”

Terraform has **dozens of built-in functions** for:

- Strings: `join()`, `split()`, `replace()`
 - Lists: `length()`, `contains()`, `distinct()`
 - Numbers: `pow()`, `log()`, `abs()`
 - Files: `file()`, `filebase64()`, etc.
-

Real-World Usage Examples

1. Dynamic Count with Logical & Comparison

hcl
CopyEdit

```
resource "aws_instance" "example" {
  count = var.environment == "prod" && var.enabled ? 1 : 0
}
```

2. Cost Calculation with Math

hcl
CopyEdit

```
locals {
  total_cost = var.instance_count * var.price_per_instance
}
```

3. Use Expressions in Conditions

hcl

CopyEdit

```
locals {  
    use_backup = var.region != "us-east-1"  
}
```

Summary

Expression Type	Operator(s)	Returns	Example
Math	+, -, *, /, %	Number	2 * 3 = 6
Equality	==, !=	Boolean	3 == 3
Comparison	>, >=, <, <=	Boolean	5 < 4
Logical	!, &&, `		

You Now Know:

-  How to use all basic expressions and operators in Terraform
 -  How to define and test them using `locals` and `output`
 -  How Terraform evaluates and returns values
 -  Best practices to test logic before writing real infrastructure
-

Next Step:

Terraform **Functions Deep Dive** — with `length()`, `join()`, `lookup()`, `regex()`, `for` expressions, and more.



Terraform Lecture 2: `for` Expressions with Lists and Objects — GOD MODE NOTES



GOAL OF THIS LECTURE

Learn to:

-  Iterate over **lists** and **objects**
 -  Transform complex types using `for` expressions
 -  Apply **filters** using `if` inside `for`
 -  Output and test transformed values
 -  Differentiate between producing **lists** vs **maps**
 -  Structure everything using `locals.tf`, `variables.tf`, `terraform.tfvars`
-



FILES & STRUCTURE USED

File	Purpose
<code>variables.tf</code>	Declares input variables
<code>terraform.tfvars</code>	Provides input values
<code>locals.tf</code>	Contains <code>locals</code> with <code>for</code> expressions
<code>outputs.tf</code>	Displays values for validation

1 STEP 1: Setup a Variable List

◆ **variables.tf**

```
hcl
CopyEdit
variable "numbers_list" {
  type = list(number)
}
```

-  Declares a list of numbers as input.
-

◆ **terraform.tfvars**

```
h
CopyEdit
numbers_list = [1, 2, 3, 4, 5]
```

-  Supplies actual numbers to use in our for-expressions.
-

② STEP 2: Basic **for** expression — Doubling Numbers

◆ **locals.tf**

```
hcl
CopyEdit
locals {
  double_numbers = [for num in var.numbers_list : num * 2]
}
```

Explanation:

Part	Meaning
[...]	We are producing a list
for num in	Iterating through input list
...	
num * 2	Output value per item

Pythonic Equivalent:

```
python
CopyEdit
[num * 2 for num in [1,2,3,4,5]]
```

 Output:

```
json
CopyEdit
double_numbers = [2, 4, 6, 8, 10]
```

3 STEP 3: Filter with `if` — Get Only Even Numbers

```
hcl
CopyEdit
locals {
    even_numbers = [for num in var.numbers_list : num if num % 2 == 0]
}
```

 Explanation:

Part	Meaning
<code>if num % 2 == 0</code>	Filter: append only even nums

 Only items where the condition is true are included.

 Output:

```
json
CopyEdit
even_numbers = [2, 4]
```

 Key Concept:

The filter happens **after** evaluating the iterable, but **before** final output inclusion.

4 STEP 4: Add Outputs to See Results

- ◆ `outputs.tf`

```
hcl
CopyEdit
output "double_numbers" {
    value = local.double_numbers
}

output "even_numbers" {
    value = local.even_numbers
}
```

5 STEP 5: Run Terraform

```
bash
CopyEdit
terraform fmt      # auto-format HCL
terraform init      # init Terraform project
terraform plan      # see expression outputs
```

Output:

```
hcl
CopyEdit
Outputs:
```

```
double_numbers = [
    2,
    4,
    6,
    8,
    10,
]
```

```
even_numbers = [
    2,
    4,
]
```

6 STEP 6: Use **for** with List of Objects

- ◆ **variables.tf — Add a List of Objects**

```
hcl
CopyEdit
variable "objects_list" {
  type = list(object({
    first_name = string
    last_name  = string
  }))
}
```

- ◆ **terraform.tfvars — Provide List of People**

```
hcl
CopyEdit
objects_list = [
  {
    first_name = "John"
    last_name  = "Smith"
  },
  {
    first_name = "Jane"
    last_name  = "Doe"
  },
  {
    first_name = "Lauro"
    last_name  = "Müller"
  }
]
```

7 STEP 7: Extract & Transform Using for

- ◆ **Get Only First Names**

```
hcl
CopyEdit
locals {
  first_names = [for person in var.objects_list : person.first_name]
```

```
}
```

 Output:

```
json
CopyEdit
first_names = ["John", "Jane", "Lauro"]
```

◆ Get Full Names (Concatenate First + Last)

```
hcl
CopyEdit
locals {
    full_names = [for person in var.objects_list :
"${person.first_name} ${person.last_name}"]
}
```

 Output:

```
json
CopyEdit
full_names = ["John Smith", "Jane Doe", "Lauro Müller"]
```

Add to outputs.tf

```
hcl
CopyEdit
output "first_names" {
    value = local.first_names
}

output "full_names" {
    value = local.full_names
}
```

Final Plan Output

```
hcl
CopyEdit
Outputs:
```

```
first_names = [  
    "John",  
    "Jane",  
    "Lauro",  
]
```

```
full_names = [  
    "John Smith",  
    "Jane Doe",  
    "Lauro Müller",  
]
```



Conceptual Deep Dive

🔁 for Expression Structure

hcl

CopyEdit

```
[for <item> in <list> : <expression> if <optional condition>]
```

Element	Role
item	Temp var for each element
list	Source list or map
expression	Transformation logic
if condition	Optional filter to exclude items

✓ You Can Also:

Source	Output	Example
e		
List	List	[for x in list : x*2]
List	Map	{for x in list : x => x*2}

Map List [for k, v in map :
 "\${k}-\${v}"]

Map Map {for k, v in map :
 upper(k) => v}



Real-World Use Cases

Use Case	Example
Extract subfields	Extract <code>id</code> , <code>arn</code> from resource lists
Filtering	Only resources with <code>status == "active"</code>
Nested loops	Flatten nested data into flat list
Transformation	Convert list of objects into map by key



Common Mistakes

Mistake	Explanation
Using <code>=</code> instead of <code>:</code>	Terraform uses colon (<code>:</code>) in <code>for</code>
Forgetting <code>[]</code> or <code>{}</code>	Determines whether you return a list or map
Invalid <code>if</code> position	Must be placed after <code>:</code>
Output wrong type	Result must match expected type of usage



Final Summary Cheatsheet

Feature	Example	Output Type
Basic transform	[for x in list : x * 2]	list
Filter values	[for x in list : x if x % 2 == 0]	list
From object list	[for p in people : p.first_name]	list

Combine fields [for p in people : "\${p.first}
\${p.last}"]

Produce map {for x in list : x => x * 2} map

What You Now Know

-  How to use `for` expressions with lists and objects
-  How to filter using `if`
-  How to output transformed results
-  How to manipulate Terraform data structures
-  That this is one of the **most powerful and common patterns** in real-world Terraform

Terraform Lecture 3: `for` Expressions with MAPS — GOD MODE NOTES

What You'll Learn in This Lecture

-  How to use `for` expressions to **iterate over maps**
 -  How to transform maps into new maps (key/value transformations)
 -  How to **filter** maps inside the loop
 -  How to build maps **from scratch** inside `locals`
 -  Differences between working with `for` on **maps vs lists**
 -  What happens with **duplicate keys** and how Terraform handles it
-

Files Used

File	Purpose
<code>variables.tf</code>	Define input map variable
<code>terraform.tfvars</code>	Provide map input values
<code>for-maps.tf</code>	Logic to transform the map
<code>outputs.tf</code>	Output the final maps

1 STEP 1: Define a Map Variable

♦ `variables.tf`

```
hcl
CopyEdit
variable "numbers_map" {
  type = map(number)
```

```
}
```

🔍 Explanation:

- Declares a **map** where the keys are strings and the values are numbers.
 - In Terraform, all map keys are strings by default.
-

◆ **terraform.tfvars**

```
h
CopyEdit
numbers_map = {
    "one" = 1,
    "two" = 2,
    "three" = 3,
    "four" = 4,
    "five" = 5
}
```

✓ Key-value structure: "key" = numeric_value.

2 STEP 2: Transforming the Map — Basic **for** Expression

◆ **for-maps.tf**

```
hcl
CopyEdit
locals {
  doubles_map = {
    for key, value in var.numbers_map :
      key => value * 2
  }
}
```

🔍 Explanation:

Part	Meaning
{ ... }	Indicates you are creating a map
for key, value in map	Iterates over key-value pairs of the map
key => value * 2	New key-value pair in generated map

✍ Output:

```
hcl
CopyEdit
doubles_map = {
  "one"    = 2,
  "two"    = 4,
  "three"  = 6,
  "four"   = 8,
  "five"   = 10
}
```

🧠 Map Creation vs List Creation (Conceptual Reminder)

Action	Terraform Syntax	Output Type
Iterate over list → create list	[for x in list : x * 2]	list
Iterate over map → create map	{for k, v in map : k => v * 2}	map
Iterate over map → create list	[for k, v in map : v * 2]	list

3 STEP 3: Filtering Map While Creating It

✚ Add Even Map

```
hcl
CopyEdit
locals {
  even_map = {
```

```
    for key, value in var.numbers_map :
        key => value * 2
        if value % 2 == 0
    }
}
```

Explanation:

- `if` clause filters the map based on value being even.
- Only entries where `value % 2 == 0` are included.

Output:

```
hcl
CopyEdit
even_map = {
    "two"  = 4,
    "four" = 8
}
```

4 STEP 4: Outputs

♦ `outputs.tf`

```
hcl
CopyEdit
output "doubles_map" {
    value = local.doubles_map
}

output "even_map" {
    value = local.even_map
}
```

5 STEP 5: Format, Plan, and Test

Terminal Commands

```
bash
CopyEdit
terraform fmt          # Format your .tf files
terraform init          # Initialize project (if not already)
terraform plan          # Preview the result
```

Expected Output:

```
hcl
CopyEdit
Outputs:
```

```
doubles_map = {
    "five"   = 10,
    "four"   = 8,
    "one"    = 2,
    "three"  = 6,
    "two"    = 4
}
```

```
even_map = {
    "four"   = 8,
    "two"    = 4
}
```

 **Note:** Order is not guaranteed in maps — keys are unordered (might appear sorted alphabetically or randomly).

What Happens with Duplicate Keys?

Terraform throws an error like:

```
kotlin
CopyEdit
Duplicate object key. Two different items produced a key
"duplicated" in this 'for' expression.
```

 **Rule:** Keys in maps must be **unique**.

 If duplicates are expected, use `...` to group them (explored in later lectures).



Advanced Key Calculation (Optional Concept)

hcl
CopyEdit
`locals {
 custom_key_map = {
 for k, v in var.numbers_map :
 "${k}_v${v}" => v * 10
 }
}`

You can compute **new keys** dynamically as long as they're unique.

💬 Conceptual Summary: `for` with Maps

Feature	Behavior
Iterate over map	Get both <code>key</code> and <code>value</code>
Create map from loop	Use <code>{}</code> and <code>key => value</code> syntax
Filter map during loop	Add <code>if</code> at the end
Key duplication	Will cause an error unless grouped
Output of <code>for</code>	Matches the bracket type (<code>[]</code> vs <code>{ }</code>)



Visual Cheatsheet

hcl
CopyEdit
`locals {
 # Map → Map
 transformed = {
 for k, v in some_map :
 k => v * 10
 }
}`

```

# Map → List
only_values = [
  for k, v in some_map : v
]

# Map → Filtered Map
evens = {
  for k, v in some_map :
    k => v if v % 2 == 0
}
}

```

🔥 Real-World Use Cases

Use Case	Example
Tag values transformation	Convert <code>map("env", "prod")</code> to uppercased keys
Filter sensitive keys	Remove keys like <code>password</code> , <code>secret</code>
Compute derived values	$\text{CPU} \times \text{cost per unit}$
API response manipulation	Build custom JSON from a map
Create dynamic Terraform inputs	Generate nested input structures

❗ Gotchas to Watch Out For

Mistake	Why It Breaks
Using <code>=</code> instead of <code>=></code> in map creation	Terraform uses <code>=></code>
Duplicated keys	Causes runtime error
Assuming map order	Maps are unordered
Forgetting to use <code>{}</code> or <code>[]</code>	Output type becomes invalid

✅ What You Now Know

- How to write `for` expressions over maps
 - How to output transformed maps
 - How to filter values during map creation
 - How Terraform handles key uniqueness
 - Difference between list and map iteration logic
-



Recap: `for` Expression Syntax Patterns

Expression Goal	Syntax Example	Output Type
Transform Map	<code>{for k, v in map : k => v * 10}</code>	Map
Filter Map	<code>{for k, v in map : k => v if v % 2 == 0}</code>	Map
Extract Map Values	<code>[for k, v in map : v]</code>	List
Extract Map Keys	<code>[for k, _ in map : k]</code>	List
Create Custom Keys	<code>{for k, v in map : "\${k}_id" => v}</code>	Map



Next Up

In the next lecture, you'll likely dive into:

- `for` expressions with **nested objects**
- Grouping using `...`
- Flattening data structures
- Real DevOps-style dynamic variable building

Terraform Lecture 4: Mapping Between Lists and Maps — GOD MODE NOTES

What This Lecture Covers

-  Converting a **list of objects** → **map**
 -  Grouping by key using `...` (ellipsis) to handle duplicate keys
 -  Creating **nested maps or object structures**
 -  Accessing values dynamically using square bracket syntax (`["key"]`)
 -  Extracting lists from maps (map → list)
 -  Making map access dynamic using variables
-

Files Involved

File Name	Purpose
<code>variables.tf</code>	Define inputs: <code>users</code> (list of objects), <code>user_to_output</code>
<code>terraform.tfv</code>	Provide actual values for the variables
<code>ars</code>	
<code>lists-maps.tf</code>	Main logic to transform structures
<code>outputs.tf</code>	Display transformed outputs

Input Variable: `users` (list of objects)

- ♦ `variables.tf`

`hcl`

CopyEdit

```
variable "users" {
  type = list(object({
    username = string
    role     = string
  }))
}
```

◆ **terraform.tfvars**

hcl
CopyEdit

```
users = [
  { username = "john", role = "admin" },
  { username = "jane", role = "developer" },
  { username = "laura", role = "auditor" },
  { username = "jane", role = "auditor" } # Duplicate username
]
```

🔍 Jane now has 2 roles: we'll **group her roles** using

🔁 STEP 1: List → Map (username as key, role as value)

◆ **lists-maps.tf**

hcl
CopyEdit

```
locals {
  users_map = {
    for user in var.users :
      user.username => user.role...
  }
}
```

🔍 Explanation:

Part	Purpose
{}	Resulting structure is a map

```
for user in var.users Iterates over list of objects  
user.username => Creates key-value pair in map  
user.role  
... after value Groups multiple values with same key into list
```

 Output:

```
hcl  
CopyEdit  
users_map = {  
    "john"  = ["admin"]  
    "jane"  = ["developer", "auditor"]  
    "laura" = ["auditor"]  
}
```

STEP 2: Map → Nested Map (Wrap value in object with roles key)

```
hcl  
CopyEdit  
locals {  
    users_map_2 = {  
        for username, roles in local.users_map :  
            username => {  
                roles = roles  
            }  
    }  
}
```

 Explanation:

- Iterates over previously created `users_map`
- Wraps each key's value (list of roles) in an object: `{ roles = [...] }`

 Output:

```
hcl  
CopyEdit
```

```
users_map_2 = {
  "john"  = { roles = ["admin"] }
  "jane"  = { roles = ["developer", "auditor"] }
  "laura" = { roles = ["auditor"] }
}
```

Outputs So Far

◆ `outputs.tf`

```
h
CopyEdit
output "users_map" {
  value = local.users_map
}

output "users_map_2" {
  value = local.users_map_2
}
```

Accessing a Specific User's Roles

Let's output `jane`'s roles from `users_map_2`.

```
hcl
CopyEdit
output "jane_roles" {
  value = local.users_map_2["jane"].roles
}
```

Output:

```
hcl
CopyEdit
jane_roles = ["developer", "auditor"]
```

Make Access Dynamic with Variable

◆ **variables.tf**

```
h  
CopyEdit  
variable "user_to_output" {  
    type = string  
}
```

◆ **terraform.tfvars**

```
hcl  
CopyEdit  
user_to_output = "jane"
```

◆ **lists-maps.tf Continued**

```
h  
CopyEdit  
output "user_to_output_roles" {  
    value = local.users_map_2[var.user_to_output].roles  
}
```

📝 Output (dynamically selected user):

```
hcl  
CopyEdit  
user_to_output_roles = ["developer", "auditor"]
```

STEP 3: Map → List (Extract usernames)

```
hcl  
CopyEdit  
locals {  
    usernames_from_map = [  
        for username, _ in local.users_map :  
            username  
    ]  
}
```

- `_` is used to ignore the value, only extract keys (usernames)

 Output:

```
hcl
CopyEdit
usernames_from_map = ["john", "jane", "laura"]
```

Output the List of Usernames

```
hcl
CopyEdit
output "usernames_from_map" {
  value = local.usernames_from_map
}
```

Summary Table: Mapping Patterns

Input Type	Output Type	Terraform Syntax
List → Map	Map	<code>{ for u in list : u.key => u.value }</code>
List → Grouped Map	Map	<code>{ for u in list : u.key => u.value... }</code>
Map → Map	Map	<code>{ for k, v in map : k => v * 2 }</code>
Map → List	List	<code>[for k, v in map : k] or [for k, v in map : v]</code>
Map → Object Map	Map	<code>{ for k, v in map : k => { roles = v } }</code>

Best Practices & Gotchas

Practice	Why It Matters
Use <code>...</code> for grouping	Avoid duplicate key errors

Use square brackets for dynamic key access	Enable runtime flexibility
Separate nested transformations	Avoid complexity and increase readability
Prefer <code>username => { ... }</code> over direct map if nesting is required	Better structure, easier access
Use <code>terraform fmt</code> regularly	Keeps code clean and aligned

Real-World Use Cases

Scenario	How You'd Use It
IAM user management	Username → role map
Grouping EC2 tags by environment	Environment → list of tags
Kubernetes Helm values	Component → values object
Dynamic outputs for modules	Convert maps to lists for dynamic <code>for_each</code>
Audit trail / role-based access tracking	Map of user → list of access scopes

What You Now Know

-  How to **convert a list of objects to a map**
-  How to **group duplicate keys into arrays** using `...`
-  How to wrap map values into **nested objects**
-  How to dynamically **access map entries using variables**
-  How to **extract lists** from maps using `for`
-  How to structure your Terraform logic clearly

Terraform Lecture 5: Splat Expressions

— GOD MODE NOTES

What You'll Learn

-  What are **splat expressions**
 -  When and how to use splats (`[*]`)
 -  How splats simplify list/object extraction
 -  Why **splats don't work with maps**
 -  Alternatives to splat expressions using `for`
 -  Using `values()` function + splat
 -  Applying splats to sets
-

What Is a Splat Expression?

A **splat expression** is a shorthand for mapping over a **list of objects** to extract a single field from each object.

```
hcl
CopyEdit
list[*].property
```

It is syntactic sugar for this:

```
hcl
CopyEdit
[for item in list : item.property]
```

File: `split.tf`

We define our logic and outputs here.



Data Source: `objects_list` (used in earlier lecture)

```
hcl
CopyEdit
variable "objects_list" {
  type = list(object({
    first_name = string
    last_name  = string
  }))
}
```

```
hcl
CopyEdit
objects_list = [
  { first_name = "John", last_name = "Doe" },
  { first_name = "Jane", last_name = "Smith" },
  { first_name = "Laura", last_name = "Stone" }
]
```



Example 1: Use Splat to Extract All First Names

```
hcl
CopyEdit
locals {
  first_names_from_splat = var.objects_list[*].first_name
}
```



Output

```
hcl
CopyEdit
output "first_names_from_splat" {
  value = local.first_names_from_splat
}
```



Result:

```
hcl
```

CopyEdit

```
first_names_from_splat = ["John", "Jane", "Laura"]
```

⚠ Splat Does NOT Work With Maps

hcl

CopyEdit

```
local.users_map_2 = {
    "john"  = { roles = ["admin"] }
    "jane"  = { roles = ["developer", "auditor"] }
    "laura" = { roles = ["auditor"] }
}
```

```
# ❌ THIS FAILS
```

```
locals {
    roles_from_splat = local.users_map_2[*].roles
}
```

🔴 Terraform Error:

typescript

CopyEdit

```
| Error: Unsupported attribute
|
| This value does not have any attributes.
```

❗ Why? Because splat expressions only work with **list-like** structures — not maps.

✓ Alternative: Use a **for** loop to Extract Roles from a Map

hcl

CopyEdit

```
locals {
    roles_from_loop = [
        for username, props in local.users_map_2 :
            props.roles
    ]
}
```



hcl

CopyEdit

```
roles_from_loop = [ ["admin"], ["developer", "auditor"], ["auditor"] ]
```

✓ Advanced: Use `values()` + Splat on Roles

hcl

CopyEdit

```
locals {  
    roles_from_splat_with_values = values(local.users_map_2)[*].roles  
}
```



Function What it does

`values(ma p)` Returns a list of values from the map

`[*].roles` Then splat accesses `roles` from each value



hcl

CopyEdit

```
roles_from_splat_with_values = [ ["admin"], ["developer", "auditor"], ["auditor"] ]
```

⌚ Alternate Naming for Simplicity

Terraform lets you break long lines across multiple lines:

hcl

CopyEdit

```
locals {
```

```
    roles_from_splat_with_values = values(local.users_map_2)[*].roles
```

```
}
```

or assign intermediate results to keep things cleaner:

```
hcl
CopyEdit
locals {
  user_values = values(local.users_map_2)
  roles_list  = local.user_values[*].roles
}
```



Output Section

```
hcl
CopyEdit
output "roles_from_loop" {
  value = local.roles_from_loop
}

output "roles_from_splat_with_values" {
  value = local.roles_from_splat_with_values
}
```



Using Splat on a **set** (not just lists!)

```
hcl
CopyEdit
variable "az_list" {
  type = set(string)
  default = ["us-east-1a", "us-east-1b", "us-east-1c"]
}

locals {
  upper_azs = var.az_list[*].upper()
}
```

Yes, `[*]` also works with **sets** because Terraform treats them similarly to lists when splatting.

Summary of Key Rules

 Works With  Doesn't Work With

List of objects Maps

Set of objects Single object

`values(map)` Direct maps

Real-World Splat Use Cases

Use Case	Example
Extract AZs	<code>data.aws_availability_zones.names[*]</code>
Get subnet IDs	<code>module.subnets[*].id</code>
Get EC2 instance IPs	<code>aws_instance.app[*].public_ip</code>
Get user emails from a list	<code>var.users[*].email</code>
Roles from IAM user map	<code>values(local.user_roles)[*].role</code>

Final Thoughts

-  Use splat to **simplify list extraction**
-  Don't use splat with maps — **it fails silently or errors**
-  Use `for` expressions when you need **flexibility or map access**
-  Use `values()` when you want to convert a **map → list**
-  Format code with `terraform fmt` regularly



Terraform Lecture 6: Terraform Functions — GOD MODE NOTES



Goal of the Lecture

To introduce Terraform's built-in **function system** and teach:

- Where to find official documentation
 - What types of functions exist
 - How to read function docs
 - When and why to use functions
 - Key gotchas (e.g. no user-defined functions)
 - Examples for common function categories
-



Where to Find Function Docs



Official Docs:

<https://developer.hashicorp.com/terraform/language/functions>



Path:

[Terraform > Language > Expressions > Functions](#)

The page includes:

- Full list of function categories
 - Descriptions
 - Usage examples
-



Function Categories in Terraform

Terraform provides **many built-in functions**, grouped into the following categories:

1. Numeric Functions

Operate on numbers

Function	Description
<code>abs()</code>	Absolute value
<code>ceil()</code>	Rounds up
<code>floor()</code>	Rounds down
<code>log()</code>	Logarithm
<code>max() / min()</code>	Extremes
<code>pow()</code>	Power

2. String Functions

Work with strings

Function	Description
<code>upper() / lower()</code>	Change case
<code>trimspace()</code>	Remove spaces
<code>replace()</code>	Replace text
<code>substr()</code>	Slice strings
<code>length()</code>	String length

3. Collection Functions

Operate on lists, sets, tuples, maps

Function	Description
----------	-------------

```
merge()      Merge maps
concat()     Combine lists
flatten()    Remove nested lists
zipmap()     Create a map from keys and
             values
chunklist   Break list into sublists
()
index()      Find index of an item
```

📌 Example — `merge()`

```
hcl
CopyEdit
merge(
  { a = "x", c = "y" },
  { c = "z", e = "w" }
)
# => { a = "x", c = "z", e = "w" }
```

Last key takes **precedence**.

4. 📁 Filesystem Functions

Function	Description
<code>file()</code>	Read file content
<code>filebase64()</code>	Base64 of file
<code>fileset()</code>	Match all files (wildcard)
<code>templatefile()</code>	Interpolate templates

5. 📅 Date & Time Functions

Function	Description
----------	-------------

`timestamp()` Current UTC timestamp

`timeadd()` Add duration

`formatdate()` Format a time

6. Hash & Crypto Functions

Function	Description
<code>md5() / sha1() / sha256()</code>	Hash strings
<code>base64encode() / decode()</code>	Encode/decode
<code>bcrypt()</code>	Secure hash

 Used for:

- Detecting file changes
- Secure configs
- Triggers

7. IP Network Functions

Function	Description
<code>cidrhost()</code>	Get IP from CIDR
<code>cidrsubnet()</code>	Subnetting
<code>cidrnetmask()</code>	Get netmask

8. Type Conversion Functions

Function	Description
<code>tostring()</code> / <code>tonumber()</code>	Convert types
<code>tolist()</code> / <code>tomap()</code>	Cast structures

 Terraform **tries to auto-convert types**, but sometimes you'll need to **cast explicitly** using these.

9. Validation Helper: `can()`

Function	Description
<code>can()</code>	Returns true if an expression does not error

 Useful for writing **custom variable validations**.

```
hcl
CopyEdit
validation {
  condition      = can(regex("^dev-", var.name))
  error_message = "Name must start with 'dev-'"
}
```

10. Encoding/Decoding Functions

Function	Description
<code>jsonencode()</code> / <code>jsondecode()</code>	Work with JSON
<code>yamldecode()</code>	Decode YAML
<code>base64encode()</code> / <code>decode()</code>	Work with base64

Useful in:

- APIs

- Template rendering
 - Passing configs
-

Custom Functions?

 **No**, Terraform does **not support** user-defined functions as of now.

Only way to compose logic:

- Combine built-in functions
 - Use `locals` to modularize
 - Use external tools (Python, Bash) if needed
-

Function Docs Format

Each function doc typically has:

1. **Name & short description**
 2. **Input/output behavior**
 3. **Example usage**
 4. **Edge cases and caveats**
-

Pro Tips

- Don't memorize every function — **know categories**
- Bookmark the Terraform Functions Docs
- Use `can()` and `merge()` regularly in **validation**, **defaults**, and **module config**

- Pair functions with `locals { }` for powerful composition
 - Avoid reinventing logic — use what's built in
-

Final Recap

Thing to Remember	Why It Matters
Terraform has tons of built-in functions	Saves you from writing extra logic
Docs are your friend	Learn by example
No custom functions	Use locals + combine
Focus on string, collection, and validation functions first	They're used all the time
You'll use <code>can()</code> , <code>merge()</code> , <code>concat()</code> , <code>file()</code> , <code>jsonencode()</code> repeatedly	Especially in real-world modules



Terraform Lecture 7: Working with Functions Practically — GOD MODE NOTES



Goal of Lecture

This lecture is a **hands-on exercise** to help you **apply Terraform functions** practically, learn how to:

- Use string and numeric functions
 - Read and decode YAML files
 - Access structured data from files
 - Encode Terraform objects into YAML/JSON
 - Chain functions and compose logic cleanly
-



Setup Steps (from the lecture)

1. Create folder `10-functions`
 2. Run `terraform init` (even though no AWS provider is required)
 3. Create file: `functions-examples.tf`
 4. Use `locals` instead of `variables` to define data for manipulation
 5. (Optional) Create `users.yaml` file to experiment with file functions
-



Example: Locals Declaration

[hcl](#)
[CopyEdit](#)

```
locals {
    name = "Laura miller"
    age  = 15
}
```

1 STRING FUNCTIONS

upper() and **lower()**

```
hcl
CopyEdit
output "example_upper" {
    value = upper(local.name)
}

output "example_lower" {
    value = lower(local.name)
}
```

- ✓ Output will be `LAURA MILLER` and `laura miller`
-

startswith()

```
hcl
CopyEdit
output "example_starts_with" {
    value = startswith(lower(local.name), "john")
}
```

- ✓ Returns `false` — useful for **conditionals and validations**
 - 🧠 Can be **chained** with `lower()` for case-insensitive checks
-

2 NUMERIC FUNCTIONS

Basic math

```
hcl
CopyEdit
```

```
output "age_times_two" {
  value = local.age * 2
}
```

abs() — absolute value

```
hcl
CopyEdit
locals {
  age = -15
}

output "age_abs" {
  value = abs(local.age) // 15
}
```

pow() — raise to power

```
hcl
CopyEdit
output "age_power" {
  value = pow(-15, 2) // 225
}
```

3 FILE FUNCTIONS + YAML/JSON

Step 1: Create **users.yaml**

```
yaml
CopyEdit
- name: Laurel
  group: developer
- name: John
  group: auditor
```

Use path: `path.module` to ensure Terraform reads from correct folder

file() + yamldecode()

```
hcl
CopyEdit
output "raw_yaml" {
    value = file("${path.module}/users.yaml")
}

output "decoded_yaml" {
    value = yamldecode(file("${path.module}/users.yaml"))
}
```

-
- 🚫 Raw `file()` output is a **multi-line string — not accessible**
 - ✓ Use `yamldecode()` to **convert to list of objects**

Access YAML objects

```
h
CopyEdit
output "first_user_name" {
    value = yamldecode(file("${path.module}/users.yaml"))[0].name
}
```

Splat Expression for all names

```
hcl
CopyEdit
output "all_user_names" {
    value = [for user in yamldecode(file("${path.module}/users.yaml"))
: user.name]
}
```

-
- ✓ Results in `["Laurel", "John"]`
-

4 ENCODING OBJECTS

Define Local Object

```
h
CopyEdit
locals {
```

```
my_object = {
    key1 = 10
    key2 = "my value"
}
}
```

Encode as YAML

```
hcl
CopyEdit
output "yaml_encoded_object" {
    value = yamldecode(file("${path.module}/users.yaml"))[0]
}
```

Encode as JSON

```
hcl
CopyEdit
output "json_encoded_object" {
    value = jsonencode(local.my_object)
}
```

Output:

```
json
CopyEdit
{"key1":10, "key2":"my value"}
```

COMPOSING FUNCTIONS

Functions can be composed:

```
h
CopyEdit
value = startswith(lower(local.name), "john")
```

File read + decode:

```
hcl
CopyEdit
```

```
value = yamldecode(file("${path.module}/users.yaml"))[0].name
```

Encode to YAML or JSON:

```
hcl  
CopyEdit  
value = yamlencode(local.my_object)
```

Summary: Key Concepts from Lecture

Category	Function(s)	Purpose
String functions	<code>upper()</code> , <code>lower()</code> , <code>startswith()</code>	Clean, compare, transform strings
Numeric	<code>*</code> , <code>abs()</code> , <code>pow()</code>	Math ops, useful in calculations
Files	<code>file()</code> , <code>yamldecode()</code>	Load structured config from external files
Encode	<code>yamlencode()</code> , <code>jsonencode()</code>	Save or output TF objects in common formats
Function chaining	<code>startswith(lower(...))</code> , etc	Compose logic in a clean way

Tips

- Always use `path.module` for relative file paths
- Terraform will throw "unsupported attribute" errors if you forget to decode structured data
- `yamldecode() + splat expressions` are super powerful for extracting structured file data
- Use this pattern heavily in real-world IaC:
 - Read secrets/configs
 - Generate dynamic values from external files

- Output structured diagnostics/debugging data

Creating Multiple Resources

Terraform – Creating Multiple Resources (Lecture 1)

This section introduces **how to efficiently create multiple resources** using two special meta-arguments:

- `count`
- `for_each`

These techniques **prevent code duplication** and help **scale resource creation dynamically** based on input data (like variables, maps, or sets).

Why Create Multiple Resources?

In real-world infrastructure:

- You might want **multiple EC2 instances**, **multiple S3 buckets**, or **multiple subnets**.
 - Writing one block per resource leads to code duplication.
 - Terraform provides smart constructs (`count`, `for_each`) to automate this.
-

Method 1: `count` Meta-Argument

Purpose:

Create **N number** of identical resource instances.

Rule:

`count` must be **known at plan time**, i.e., Terraform must know the number before it starts planning and applying.

Syntax:

```
h
CopyEdit
resource "aws_instance" "example" {
  count      = 3
  ami        = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "Instance-${count.index}"
  }
}
```

✓ Breakdown:

- `count = 3` → Creates **3 instances**.
- `${count.index}` → Accesses **current instance index** (starts from `0`).

📌 Accessing:

- **Single instance** → `aws_instance.example[0]`, `aws_instance.example[1]`, ...
- **All instances** → Just use `aws_instance.example` (returns list of instances)

📝 Real-world Example:

```
hcl
CopyEdit
variable "ec2_count" {
  default = 2
}

resource "aws_instance" "multiple" {
  count      = var.ec2_count
  ami        = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "MyInstance-${count.index}"
  }
}
```

```
    }  
}
```

If `var.ec2_count = 2`, you get:

- `aws_instance.multiple[0]`
 - `aws_instance.multiple[1]`
-



count Limitations & Gotchas:

Limitation	Explanation
Index-based	You must use numeric indexes, which may cause instability if order changes.
Limited flexibility	You can't assign custom names or configs easily per instance.
Must be known at plan time	Cannot use values that are only known at apply-time (like outputs of a resource).



Method 2: `for_each` Meta-Argument

✓ Purpose:

Create **multiple resources with unique configurations**, using **maps or sets of strings**.

✓ Syntax:

```
hcl  
CopyEdit  
resource "aws_subnet" "main" {  
  for_each = var.subnet_config  
  
  vpc_id      = var.vpc_id  
  cidr_block  = each.value.cidr_block  
  
  tags = {  
    Name = "Subnet-${each.key}"  
  }  
}
```

```
}
```

✓ Breakdown:

`var.subnet_config` is a **map**:

```
h  
CopyEdit  
{  
    "subnet_a" = { cidr_block = "10.0.1.0/24" }  
    "subnet_b" = { cidr_block = "10.0.2.0/24" }  
}
```

-
- `each.key` → "subnet_a", "subnet_b"
- `each.value` → { `cidr_block` = "..." }

📌 Accessing:

- `aws_subnet.main["subnet_a"]`
- `aws_subnet.main["subnet_b"]`

✍️ Real-world Example:

```
hcl  
CopyEdit  
variable "subnet_config" {  
    default = {  
        "subnet-a" = { cidr_block = "10.0.1.0/24" }  
        "subnet-b" = { cidr_block = "10.0.2.0/24" }  
    }  
}  
  
resource "aws_subnet" "main" {  
    for_each = var.subnet_config  
  
    vpc_id      = var.vpc_id  
    cidr_block = each.value.cidr_block
```

```

tags = {
    Name = each.key
}
}

```

You now get:

- `aws_subnet.main["subnet-a"]`
 - `aws_subnet.main["subnet-b"]`
-

count vs for_each – Comparison Table

Feature	<code>count</code>	<code>for_each</code>
Input type	Single integer	Map or Set of strings
Access style	<code>count.index</code>	<code>each.key / each.value</code>
Index stability	Fragile (based on order)	Stable (based on keys)
Use cases	Identical resources	Unique configs per resource
Chainability	Yes	Yes
Must be known at plan	Yes	Yes

Important Notes:

1. Don't use sensitive values in `for_each`

- Keys in `for_each` must be **stable and known**, e.g., not derived from secrets or outputs of dynamic resources.

2. You can chain resources

Use the full resource like:

```
hcl
CopyEdit
aws_subnet.main
```

And chain it into another `for_each`, like this:

```
h
CopyEdit
resource "aws_route_table_association" "assoc" {
  for_each = aws_subnet.main

  subnet_id      = each.value.id
  route_table_id = aws_route_table.main.id
}
```

•

3. 🔥 Using sets vs maps in `for_each`:

- **Maps** → `each.key, each.value`
- **Sets of strings** → `each.key == each.value` (they're the same)

✓ Summary – Key Learnings:

Topic	Key Point
<code>count</code>	Use when you want N identical resources
<code>for_each</code>	Use when each resource needs different config
Accessing instances	<code>count.index</code> vs <code>each.key / each.value</code>
Stability	<code>for_each</code> is safer than <code>count</code> (no index reordering issues)
Chaining resources	Both <code>count</code> and <code>for_each</code> can be chained into other blocks
Plan-time values	Both must be known before apply
Sensitive data	Shouldn't be used in <code>for_each</code> input



Pro Tips:

-  Use `for_each` for anything more than a repeat loop.

Even if you're just repeating "A", "B", "C" with slightly different inputs, use `for_each`.

-  Want to dynamically generate EC2s with different tags? Use `for_each`.
-  Avoid `count` when list order may change — it's dangerous because Terraform may `recreate` everything due to index shift.
-  Keep keys in `for_each` stable and predictable — think like unique IDs or names.



Terraform – Creating Multiple Resources (Lecture 2: Practical with count)



Goal of this Lecture

Use the `count` meta-argument to **dynamically create multiple similar resources**, reduce code duplication, and **parameterize the number of resources** (e.g., subnets) with a variable.



Project Structure Setup

Create Directory:

```
bash
CopyEdit
mkdir 11-multiple-resources
cd 11-multiple-resources
```



1. provider.tf – Provider Configuration

```
hcl
CopyEdit
terraform {
  required_version = "~> 1.7"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
```

```
    region = "eu-west-1" # Use any region applicable to you
}
```

 Run these in terminal:

```
bash
CopyEdit
terraform fmt
terraform init
source .env # load AWS credentials (ensure this file has your
AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY)
```



2. **variables.tf – Define Variables**

```
hcl
CopyEdit
variable "subnet_count" {
  type    = number
  default = 2
}
```



3. **networking.tf – Define Resources**

Step 1: Create a VPC

```
hcl
CopyEdit
locals {
  project = "11-multiple-resources"
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name      = local.project
    Project   = local.project
  }
}
```

Step 2: Create Subnets Using `count`

```
hcl
CopyEdit
resource "aws_subnet" "main" {
    count                  = var.subnet_count
    vpc_id                 = aws_vpc.main.id
    cidr_block              = "10.0.${count.index}.0/24"

    tags = {
        Name      = "${local.project}-subnet-${count.index}"
        Project   = local.project
    }
}
```

🔥 Explanation:

Line	What it Does
<code>count = var.subnet_count</code>	Creates as many subnets as the variable value
<code>cidr_block = "10.0.\${count.index}.0/24"</code>	Dynamically assigns non-overlapping CIDR blocks
<code>Name</code> tag includes count index	Helps visually identify resources in the AWS console

🧪 Terraform Plan & Apply

Initial Run:

```
bash
CopyEdit
terraform plan
terraform apply -auto-approve
```

✖ Error Handling – Overlapping CIDR Issue

If you **don't use** `count.index` and hardcode the CIDR block:

```
hcl
CopyEdit
cidr_block = "10.0.0.0/24"
```

You'll get this error on `terraform apply`:

 **Error:** "The CIDR blocks are overlapping"

AWS **does not allow duplicate subnet CIDRs** in the same VPC.

Fix: Use `count.index` in CIDR block

```
hcl
CopyEdit
cidr_block = "10.0.${count.index}.0/24"
```

This ensures:

- Each subnet is in a separate IP range.
 - No conflict.
 - Safe scalability.
-

AWS Console Verification

- Check **VPCs** > You should see your custom VPC.
 - Check **Subnets** > Multiple subnets with different CIDRs.
 - Tags should show names like:
 - `11-multiple-resources-subnet-0`
 - `11-multiple-resources-subnet-1`
-

Partial Resource Creation Warning

If Terraform fails mid-way (e.g., subnet conflict), **some resources might still be created**.

AWS is **eventually consistent**, so:

- It may take time for deleted resources to disappear.
- You might see "ghost" resources in the console.
- Trying to delete manually may result in “resource doesn’t exist” error.

 Just re-run:

```
bash
CopyEdit
terraform apply -auto-approve
```

Adding Flexibility with `terraform.tfvars`

 Create a file: `terraform.tfvars`

```
hcl
CopyEdit
subnet_count = 3
```

Then run:

```
bash
CopyEdit
terraform apply
```

Plan Output Will Show:

- 1 new subnet being added.
- With CIDR block `10.0.2.0/24`.

Changing Subnet Count Dynamically

Change Made	Terraform Response
Increase subnet count	Adds new subnet(s)
Decrease subnet count	Deletes excess subnets
Modify CIDR structure	Replaces subnets (forces replacement)



Understanding "Forces Replacement"

When Terraform plans changes to immutable properties (like `cidr_block`), the plan will show:

```
~ cidr_block: "10.0.0.0/24" => "10.0.1.0/24" (forces replacement)
```

⚠ This means the **subnet will be destroyed and recreated**.



Key Concepts Recap

Concept	Explanation
<code>count</code>	Used to create multiple identical resources
<code>count.index</code>	Zero-based index to identify each resource
CIDR conflict	AWS will fail if subnets have overlapping CIDRs
Terraform plan vs apply	<code>plan</code> may succeed even if <code>apply</code> will fail
Using variables	Avoids hardcoding; enables dynamic infrastructure
<code>terraform.tfvars</code>	Provides environment-specific values for variables
Tagging best practices	Use clear and indexed <code>Name</code> tags for traceability
AWS eventual consistency	UI delays may show deleted resources as "existing" for a short while



Pro Tips

- 💡 Always use `${count.index}` in CIDRs when creating multiple subnets.
 - 🍩 Clean up AWS Console after failure using `terraform destroy` if things get messy.
 - 🔄 `terraform.tfvars` is powerful for environment-specific customization (e.g., staging vs prod).
 - 🛡 Tag everything. Helps when debugging and cleaning resources.
 - 🔍 If resources don't appear immediately in AWS, wait or refresh — console is eventually consistent.
 - 🚫 Don't create multiple resources by copy-pasting – it leads to brittle code. Use `count`.
-

What You've Learned

You can now:

- Set up a project with provider and variables
- Create a **VPC**
- Use `count` to dynamically create **multiple subnets**
- Avoid **CIDR conflicts**
- Parameterize the **number of subnets** via `.tfvars`
- Understand **error handling, replacement, and eventual consistency**



Lecture 3 – Referencing Resources

Created with **count** in Terraform

This lecture expands on how to **reference resources created using `count`**, especially:

- Working with lists of resources
 - Avoiding errors when referencing by index
 - Dynamically assigning EC2s to subnets using a **round-robin strategy**
-



Folder Structure

📁 11-multiple-resources/

bash

CopyEdit

```
├── provider.tf
├── variables.tf
├── networking.tf      # Contains VPC + Subnet definitions
(already done in Lecture 2)
├── data.tf            # Shared locals
├── compute.tf         # New: EC2 instances with count
└── terraform.tfvars  # Optional variable overrides
```



Step 1: Add Variable for EC2 Count (**variables.tf**)

hcl

CopyEdit

```
variable "ec2_instance_count" {
  type    = number
  default = 1
}
```



Step 2: Define Local Project Name (**data.tf**)

```
hcl
CopyEdit
locals {
  project = "11-multiple-resources"
}
```

Step 3: Add Ubuntu AMI Data Source (inside `compute.tf`)

```
hcl
CopyEdit
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name    = "name"
    values =
    [ "ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*" ]
  }

  filter {
    name    = "virtualization-type"
    values = [ "hvm" ]
  }

  owners = [ "099720109477" ] # Canonical
}
```

Step 4: Create EC2 Instances using `count` (`compute.tf`)

```
hcl
CopyEdit
resource "aws_instance" "from_count" {
  count          = var.ec2_instance_count
  ami            = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
```

```

subnet_id = aws_subnet.main[
    count.index % length(aws_subnet.main)
].id

tags = {
    Name      = "${local.project}-instance-${count.index}"
    Project   = local.project
}
}

```

Deep Explanation of the Above Code

Concept	Explanation
count	Creates N EC2 instances, based on variable
count.index	Current EC2 instance's index (0-based)
aws_subnet.main[...]	aws_subnet.main is a list, must reference index
count.index % length(aws_subnet.main)	Round-robin strategy to cycle through subnet indexes

Round-Robin Example (2 Subnets, 4 Instances):

EC2 Index	Subnet Index = <code>index % Subnet</code>	Subnet
0	0	Subnet-0
1	1	Subnet-1
2	0	Subnet-0
3	1	Subnet-1

 This ensures **even distribution of EC2s across subnets.**

Plan and Apply

Plan:

```
bash
CopyEdit
terraform plan
```

Apply:

```
bash
CopyEdit
terraform apply -auto-approve
```

 If you get errors like:

```
perl
CopyEdit
Invalid index
```

It likely means you're trying to access a subnet index (e.g., `aws_subnet.main[2]`) that **doesn't exist**.



Console Debugging – Check Subnet ID for Each EC2

1. Go to EC2 Dashboard
 2. Modify visible columns to include `Subnet ID`
 3. Confirm:
 - Instance 0 and 2 → in Subnet 0
 - Instance 1 and 3 → in Subnet 1
-



AWS Free Tier Safety Tips

- Free tier includes **750 hours/month** of t2.micro or t3.micro

- Running 4 instances for 10 minutes = **40 minutes consumed**
- Always **destroy resources** after testing:

```
bash
CopyEdit
terraform destroy -auto-approve
```

Bonus: If you want 100% control of instance-to-subnet mapping

You can define a **list of objects** and loop through it with `for_each` instead of `count`.

Example:

```
hcl
CopyEdit
variable "instance_configs" {
  default = [
    { name = "web1", subnet_index = 0 },
    { name = "web2", subnet_index = 1 },
    { name = "web3", subnet_index = 0 },
  ]
}

resource "aws_instance" "custom" {
  for_each = { for idx, val in var.instance_configs : idx => val }

  ami          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  subnet_id    = aws_subnet.main[each.value.subnet_index].id

  tags = {
    Name      = each.value.name
    Project   = local.project
  }
}
```

Recap – Key Concepts Covered

Concept	What You Learned
<code>count.index</code>	Used to get EC2 index while creating multiple
Referencing Subnet	Must use index: <code>aws_subnet.main[0]</code>
Error Avoidance	Don't access out-of-range indexes
Round-robin logic	Use modulo: <code>count.index % length(...)</code>
Variable driven infra	Use <code>var.ec2_instance_count</code>
Free tier advice	Destroy infra after usage
Shared locals	Keep global values in <code>data.tf</code> or <code>locals.tf</code>

Pro Tips

-  Always pair `count` with `count.index` for customization
-  Use `length()` and `%` for safe round-robin logic
-  Avoid hardcoding subnet indexes — especially when number of subnets may change
-  Don't reference `.id` directly from a resource that uses `count` without index
-  Always run `terraform plan` before `apply` to avoid surprises
-  Clean up with `terraform destroy` to stay under the free tier

What You Now Know

You are now capable of:

- Referencing any resource created via `count`

- Looping over subnets and assigning EC2s intelligently
- Building scalable infra logic in Terraform
- Avoiding common indexing pitfalls
- Leveraging `locals`, `data`, `variables`, and computed values together

Lecture 4: Creating EC2 Instances from a List of Configs using `count`

Key Concept Shift from Lecture 3:

In Lecture 3, we created multiple EC2s using:

- `count = var.ec2_instance_count`
- Logic like `count.index % length(subnets)` for round-robin subnet distribution

In Lecture 4, we level up:

- Use a **list of objects** (i.e. structured configs per EC2)
 - Dynamically pull `ami` and `instance_type` from each object
 - Still use `count`, **not `for_each`**, because `for_each` does **not support list of objects**
-

Project Setup

Folder: `11-multiple-resources/`

Make sure you have these files:

```
bash
CopyEdit
.
├── provider.tf
├── data.tf
├── networking.tf
├── compute.tf
├── variables.tf
└── terraform.tfvars
```

Step-by-Step Setup

Step 1: Add a Variable for EC2 Config List (`variables.tf`)

hcl
CopyEdit

```
variable "ec2_instance_config_list" {
  type = list(object({
    instance_type = string
    ami           = string  # this will be a label like "ubuntu"
  }))
}
```

Step 2: Provide Values in `terraform.tfvars`

hcl
CopyEdit

```
ec2_instance_config_list = [
  {
    instance_type = "t2.micro"
    ami          = "ubuntu"
  },
  {
    instance_type = "t2.micro"
    ami          = "ubuntu"
  }
]
```

 We are not giving raw AMI IDs here, just **short keys** like "`ubuntu`", which we'll map later.

Step 3: Disable Previous `ec2_instance_count`

If still present in your Terraform files, set:

hcl
CopyEdit

```
ec2_instance_count = 0
```

Otherwise it will try to create two sets of EC2s.

Step 4: Add a Data Source for Ubuntu AMI ([compute.tf](#))

```
hcl
CopyEdit
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name    = "name"
    values =
    ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  filter {
    name    = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```

Step 5: Create `locals` to Map AMI Labels to AMI IDs ([data.tf](#))

```
hcl
CopyEdit
locals {
  ami_ids = {
    ubuntu = data.aws_ami.ubuntu.id
  }

  project = "11-multiple-resources"
}
```

 You can extend this later (e.g., add "nginx" or "amazonlinux").

Step 6: EC2 Resource Using `count` and `Configs` ([compute.tf](#))

```

h
CopyEdit
resource "aws_instance" "from_list" {
  count      = length(var.ec2_instance_config_list)
  instance_type =
var.ec2_instance_config_list[count.index].instance_type

  ami = local.ami_ids[
    var.ec2_instance_config_list[count.index].ami
  ]

  subnet_id = aws_subnet.main[
    count.index % length(aws_subnet.main)
  ].id

  tags = {
    Name      = "${local.project}-instance-${count.index}"
    Project   = local.project
  }
}

```

Deep Explanation

Terraform Feature	How It's Used
<code>list(object)</code> variable	Defines EC2 configs with <code>instance_type</code> and <code>ami</code> label
<code>count</code>	Iterates over list to create EC2s
<code>count.index</code>	Used to access individual object from the list
<code>local.ami_ids[...]</code>	Resolves short label like " <code>ubuntu</code> " to actual AMI ID
<code>count.index % length(subnet)</code>	Round-robin assignment of EC2s to subnets

Why You Can't Use `for_each` Here

Terraform only allows:

- `for_each` with **maps** or **sets of strings**

Your variable:

```
h
CopyEdit
list(object({ instance_type = ... , ami = ... }))
```

So this will fail:

```
hcl
CopyEdit
for_each = var.ec2_instance_config_list # ✗ Invalid
```

Error:

```
arduino
CopyEdit
Invalid for_each argument: Must be a map or set of strings
```

 Correct approach: use `count` with `length(var.list)` and access with `count.index`.



Terraform Plan Output Expectations

When you run:

```
bash
CopyEdit
terraform plan
```

You should see:

- EC2 resources named like `aws_instance.from_list[0]`, `[1]`, etc.
 - Each one with a unique instance type and AMI based on the config list
-

What Happens If You Input a Bad AMI Label?

hcl

CopyEdit

```
ami = "ubuntu2" # ❌ Not defined in local.ami_ids
```

You'll get an error like:

kotlin

CopyEdit

```
The given key does not identify an element in this collection value
```

 Always validate user inputs OR extend your `local.ami_ids` with expected keys.

Summary Table

Concept	Value
Config Driven Infra	✓
Custom EC2 per config	✓
Round-robin subnet logic	✓
Variable AMI selection	✓
<code>count</code> + list of object	✓
<code>for_each</code> with list of object	❌ Not allowed

Pro Tips

- Always use `count` for lists of objects
- Keep reusable mappings (like AMIs) inside `locals`
- Prefer **short identifiers** for AMIs (e.g., "`ubuntu`") and resolve them internally
- Don't hardcode AMI IDs in tfvars — they change over time

- Separate shared variables/locals into `data.tf` or `locals.tf`
 - Catch invalid config early using Terraform validation blocks (optional)
-

Cleanup After Running

To destroy and avoid AWS charges:

```
bash
CopyEdit
terraform destroy -auto-approve
```

Coming Next in Lecture 5

- Extend `local.ami_ids` to support more types like `nginx`
 - Make your AMI config even more flexible
-

Would you like:

- A **ready-to-run ZIP** of all the above `.tf` files?
- Or a **diagram** showing how the instances are linked to subnets + AMIs?

Let me know — I'll generate it for you!

Lecture 5: Adding Support for NGINX AMIs in EC2 Config List

Goal Recap from Lecture 4:

Previously, we used a list of objects to create multiple EC2s dynamically using:

- `instance_type` (e.g. "t2.micro")
- `ami_label` (e.g. "ubuntu")

Now we add **support for another image — `nginx`.**

Final Terraform Structure

```
graphql
CopyEdit
.
├── compute.tf      # EC2 instances
├── data.tf         # AMI data sources + locals
├── variables.tf    # List of object variable
└── terraform.tfvars # Input configs
```

Changes & Steps (Chronological)

Step 1: Add New AMI Source for Bitnami NGINX (`compute.tf` or `data.tf`)

```
hcl
CopyEdit
data "aws_ami" "nginx" {
  most_recent = true

  filter {
    name    = "name"
```

```
    values =
[ "bitnami-nginx-1.25.4-*-linux-debian-12-x86_64-hvm-ebs*" ]
}

filter {
  name    = "virtualization-type"
  values = [ "hvm" ]
}

filter {
  name    = "root-device-type"
  values = [ "ebs" ]
}

owners = [ "979382823631" ] # Bitnami official publisher
}
```

 This AMI comes from AWS Marketplace > Bitnami > NGINX Open Source

Step 2: Extend `locals.ami_ids` (`data.tf`)

```
hcl
CopyEdit
locals {
  ami_ids = {
    ubuntu = data.aws_ami.ubuntu.id
    nginx = data.aws_ami.nginx.id
  }

  project = "11-multiple-resources"
}
```

Now, if user provides "`nginx`" in the object, it will resolve properly.

Step 3: Update `terraform.tfvars`

```
hcl
CopyEdit
ec2_instance_config_list = [
```

```
{  
    instance_type = "t2.micro"  
    ami           = "ubuntu"  
,  
{  
    instance_type = "t2.micro"  
    ami           = "nginx"  
}  
}  
]
```

✓ Step 4: Keep the `aws_instance` Resource As-Is (from Lecture 4)

hcl

CopyEdit

```
resource "aws_instance" "from_list" {  
    count          = length(var.ec2_instance_config_list)  
    instance_type =  
        var.ec2_instance_config_list[count.index].instance_type  
  
    ami = local.ami_ids[  
        var.ec2_instance_config_list[count.index].ami  
    ]  
  
    subnet_id = aws_subnet.main[  
        count.index % length(aws_subnet.main)  
    ].id  
  
    tags = {  
        Name      = "${local.project}-instance-${count.index}"  
        Project   = local.project  
    }  
}
```

This **dynamically switches the AMI** based on what the user provides in the config.

⚠ If You Use an Invalid AMI Label?

h

CopyEdit

```
ami = "nginx_v2" # ❌
```

You'll get:

```
kotlin
CopyEdit
The given key does not identify an element in this collection value.
```

 Consider adding validation blocks later to restrict values.

How They Verified the NGINX AMI

1. Go to EC2 Console → Launch Instance → Browse More AMIs
2. Search for "nginx" → choose Bitnami NGINX

Note the AMI name pattern:

```
markdown
CopyEdit
bitnami-nginx-1.25.4-*-linux-debian-12-x86_64-hvm-ebs*
```

- 3.
4. Use this in `filter.values` in the data block.

 Pattern-based filters ensure we don't hardcode exact AMI IDs which change often.

Cleanup (To Avoid Charges)

```
bash
CopyEdit
terraform destroy -auto-approve
```

Always destroy EC2 instances when done to prevent accidental billing.

Summary Table

Feature	Value
AMI Selection via Label	
Dynamic mapping via <code>locals.ami_ids</code>	
Support for Ubuntu & Bitnami NGINX	
Uses <code>count</code> for object list	
Subnet round-robin logic	
Hardcoded AMI ID avoided	 (pattern-based filters)



Pro Tips

- Always generalize AMI filters with wildcards (*) for stability
- Use a `map` in `locals` for all label → ID logic to avoid string matching in the resource
- Use `terraform console` to debug AMI IDs & indexes interactively
- Tag EC2s by `count.index` for clarity when debugging
- Keep AMI filters and owners **documented** so teammates can trace their source



What's Next in Lecture 6?

We'll likely explore how to extend this further — maybe even go **modular**, or allow **dynamic tags**, **EBS volumes**, or **cloud-init** startup scripts.

Would you like:

- A **ready-made .zip** of the Terraform files for this lecture?
- A **dynamic config validator** for AMI labels?

- Or a **diagram** showing the full variable-to-resource flow?

Let me know — I'll generate it instantly.

Lecture 6: Adding Validation to list(object) Variables in Terraform



The Problem

You're accepting user-defined EC2 configurations like:

```
hcl
CopyEdit
ec2_instance_config_list = [
  {
    instance_type = "t2.micro"
    ami           = "ubuntu"
  },
  {
    instance_type = "t3.micro" # ✗
    ami           = "nginx2"   # ✗
  }
]
```

Without validation:

- `t3.micro` would be **silently deployed** (or break only during AWS deploy).
 - "`nginx2`" would break with a **cryptic Terraform index error**.
-



The Goal

Prevent invalid configurations before they cause runtime or deploy errors, using `validation` blocks in `variables.tf`.

- ✓ Block `instance_type` if it's not "`t2.micro`"
 - ✓ Block `ami` if it's not "`ubuntu`" or "`nginx`"
 - ✓ Give **friendly, helpful error messages**
-

Variable Definition Recap

```
hcl
CopyEdit
variable "ec2_instance_config_list" {
  type = list(object({
    instance_type = string
    ami           = string
  }))
}
```

Validation 1: Allow *only* t2.micro

Logic Overview

We:

1. **Loop** over all elements in the object list
2. **Check** that all `.instance_type` fields equal "t2.micro"
3. **Wrap** in `alltrue()` to ensure *every* entry passes

Code:

```
hcl
CopyEdit
validation {
  condition = alltrue([
    for config in var.ec2_instance_config_list :
      contains(["t2.micro"], config.instance_type)
  ])
  error_message = "Only \"t2.micro\" instance types are allowed."
}
```

 Use `contains()` so it's easy to extend to other allowed types later (`["t2.micro", "t3.micro"]` etc).

Validation 2: Allow *only* "ubuntu" or "nginx" for ami

Code:

```
hcl
CopyEdit
validation {
    condition = alltrue([
        for config in var.ec2_instance_config_list :
            contains(["ubuntu", "nginx"], config.ami)
    ])
    error_message = <><EOT
At least one of the provided AMI values is not supported.
Supported AMI values: "ubuntu", "nginx".
EOT
}
```

 Much cleaner than the cryptic “invalid index for object” error Terraform gives by default

What Happens on Failure?

Example:

```
hcl
CopyEdit
ec2_instance_config_list = [
    {
        instance_type = "t3.micro"
        ami           = "nginx2"
    }
]
```

Terraform Plan Output:

```
pgsql
CopyEdit
|
| Error: Invalid value for variable
```

```
|   The provided instance_type is not allowed.  
|   Only "t2.micro" instance types are allowed.  
  
|   The provided AMI value is not supported.  
|   Supported AMI values: "ubuntu", "nginx".
```

 Clear. Friendly. Fixable.

Dev Experience Matters

"The first user of your Terraform module is you. The second is future-you. The third is a sleepy intern at 3AM."

✓ These validations:

- Prevent frustrating debug time
 - Help users follow best practices
 - Make your code production-ready and self-documenting
-

Terraform Functions Used

Function	Purpose
<code>for</code>	Loop over list of objects
<code>contains</code>	Check if value is in allowed list
<code>()</code>	
<code>alltrue(</code>	Ensure every check passes (bool[])
<code>)</code>	
<code>validati</code>	Custom check inside variable block
<code>on</code>	

Suggested Enhancements

- Add `description` in each `variable` block for better IDE experience
 - Write test `.tfvars` for good and bad inputs to test validation quickly
 - Use `precondition` (Terraform 1.3+) inside `resource` blocks for deeper-level validation
 - Consider a **module wrapper** that restricts these values centrally
-

Final `variables.tf` Snippet

hcl

CopyEdit

```
variable "ec2_instance_config_list" {
    description = "List of EC2 instance configurations with
instance_type and ami"

    type = list(object({
        instance_type = string
        ami           = string
    }))
}

validation {
    condition = alltrue([
        for config in var.ec2_instance_config_list :
        contains(["t2.micro"], config.instance_type)
    ])
    error_message = "Only \"t2.micro\" instance types are allowed."
}

validation {
    condition = alltrue([
        for config in var.ec2_instance_config_list :
        contains(["ubuntu", "nginx"], config.ami)
    ])
    error_message = <<EOT
At least one of the provided AMI values is not supported.
Supported AMI values: "ubuntu", "nginx".
EOT
}
```

}

✓ Summary

Aspect	✓ Done
Validates EC2 instance type	✓ Only allows "t2.micro"
Validates AMI label	✓ "ubuntu" or "nginx" only
Helpful error messages	✓ Human-readable
Terraform-safe	✓ No runtime failure

🧠 Real World Wisdom



Terraform is **declarative**, but validation makes it **defensive**.

- Good Terraform isn't just about creating infra
- It's about creating **safe, understandable, and reusable** infra



Lecture 7: Using `map(object)` Instead of `list(object)` for Resource Creation in Terraform



The Problem With Lists

When using `list(object)` to create multiple EC2 instances:

```
hcl
CopyEdit
ec2_instance_config_list = [
  { ami = "ubuntu", instance_type = "t2.micro" },
  { ami = "nginx", instance_type = "t2.micro" }
]
```

If you **swap the order** of items, like:

```
hcl
CopyEdit
ec2_instance_config_list = [
  { ami = "nginx", instance_type = "t2.micro" },
  { ami = "ubuntu", instance_type = "t2.micro" }
]
```

Terraform:

- **Destroys and recreates** both instances
- Thinks the AMI has changed, because it tracks by **index (0, 1, 2...)**, not value

This leads to **unnecessary infra churn** — even if the configs remain logically the same.



The Solution: Use a `map(object)`

Instead of relying on order, give each resource a **stable key**.

Example:

```
hcl
CopyEdit
ec2_instance_config_map = {
  "ubuntu1" = {
    ami          = "ubuntu",
    instance_type = "t2.micro",
    subnet_index  = 0
  },
  "nginx1" = {
    ami          = "nginx",
    instance_type = "t2.micro",
    subnet_index  = 1
  }
}
```

This:

- Eliminates problems caused by reordering
 - Provides **semantic names** for resources (e.g. "nginx1")
 - Improves maintainability in dynamic or generated setups
-



How Terraform Handles Maps with `for_each`

When you do:

```
hcl
CopyEdit
resource "aws_instance" "from_map" {
  for_each = var.ec2_instance_config_map
}
```

Terraform gives you:

- `each.key` = the map key (e.g. "nginx1")
- `each.value` = the full object for that key

Converting From `list` to `map`

Step-by-Step

1. Change the variable type

```
hcl
CopyEdit
variable "ec2_instance_config_map" {
  type = map(object({
    ami          = string
    instance_type = string
    subnet_index  = optional(number, 0)
  }))
}
```

2. Update the `.tfvars`

```
h
CopyEdit
ec2_instance_config_map = {
  "ubuntu1" = {
    ami          = "ubuntu"
    instance_type = "t2.micro"
  },
  "nginx1" = {
    ami          = "nginx"
    instance_type = "t2.micro"
    subnet_index  = 1
  }
}
```

3. Update the resource

```
hcl
CopyEdit
resource "aws_instance" "from_map" {
```

```
for_each = var.ec2_instance_config_map

ami          = local.ami_ids[each.value.ami]
instance_type = each.value.instance_type
subnet_id    = aws_subnet.main[each.value.subnet_index].id
tags = {
  Name = each.key
}
}
```

✓ No more `count.index` — use `each.key` and `each.value` instead

📌 Indexing Behavior Comparison

List-based (`count`)

Indexed by position (0,1,2...)

Reordering = resource
replace

Hard to track externally

Map-based (`for_each`)

Indexed by key (e.g., "web1")

Reordering = no change

Easy to identify by key



Terraform Plan Behavior

When switching from list → map:

- Terraform **destroys** resources from the list
- Terraform **adds** new resources from the map

💡 You can't directly rename them across — **moved blocks** (covered later) can help retain state.



Output Snapshot

shell

Copy>Edit

Terraform will perform the following actions:

```
# aws_instance.from_list[0] will be destroyed
# aws_instance.from_list[1] will be destroyed
# aws_instance.from_map["ubuntu1"] will be created
# aws_instance.from_map["nginx1"] will be created
```

Final Thoughts

Principle	Summary
Avoid list for dynamic infra	Use <code>map(object) + for_each</code>
Make state stable	Key-based indexing avoids unnecessary replacements
Future-proofing	Easier to integrate with automation (e.g. auto-generating keys)
Readable tagging	<code>each.key</code> makes it easier to trace in AWS console

Terraform Functions & Concepts

Function / Concept	Purpose
<code>for_each</code>	Iterates over maps (or sets)
<code>each.key</code>	The string key from map
<code>each.value</code>	The full object per map entry
<code>optional()</code>	Define optional field in object
<code>map(object(...))</code>	Strong typing and safety
<code>)</code>	

Final `variables.tf` Snippet

```
hcl
CopyEdit
variable "ec2_instance_config_map" {
  description = "Map of EC2 instances with their configs"
```

```
type = map(object({
    instance_type = string
    ami           = string
    subnet_index = optional(number, 0)
})))
}
```

When to Use List vs Map

Use Case	Recommended Type
Stable, named instances	 map(object)
Dynamic count, flexible naming	 list(object)
Indexed by numeric position	 count

Bonus Pro Tip

To migrate an old list-based resource to map **without triggering a destroy**, you can use:

```
hcl
CopyEdit
moved {
  from = aws_instance.from_list[0]
  to   = aws_instance.from_map["nginx1"]
}
```

(This will be covered in future lectures.)



Lecture 8: Adding Validation to `map(object)` Variables in Terraform



Goal

Apply the **same validations** we had on our `list(object)` to our new `map(object)` setup for EC2 instances.



What We're Validating

Field	Validation Rule
<code>instance_type</code>	Must be only " <code>t2.micro</code> "
<code>e</code>	
<code>ami</code>	Must be either " <code>ubuntu</code> " or " <code>nginx</code> "
 <code>subnet_index</code>	Cannot be validated against existing resources inside a variable block (shown why)



Understanding How to Loop Over Map Values

Two Valid Syntax Options

1. Using `for key, value in:`

```
hcl
CopyEdit
[for key, config in var.ec2_instance_config_map :
contains(["t2.micro"], config.instance_type)]
```

2. Using `values()` function:

```
hcl
CopyEdit
```

```
[for config in values(var.ec2_instance_config_map) :  
contains(["t2.micro"], config.instance_type)]
```

- Both give you a list of booleans which you can pass into `alltrue()` to ensure all entries are valid.
-

✓ Final Validation Block for `instance_type`

h
CopyEdit

```
validation {  
    condition = alltrue([  
        for config in values(var.ec2_instance_config_map) :  
            contains(["t2.micro"], config.instance_type)  
    ])  
    error_message = "Only 't2.micro' instance types are allowed."  
}
```

✓ Final Validation Block for `ami`

hcl
CopyEdit

```
validation {  
    condition = alltrue([  
        for config in values(var.ec2_instance_config_map) :  
            contains(["ubuntu", "nginx"], config.ami)  
    ])  
    error_message = <<EOT  
At least one of the provided AMI values is not supported.  
Supported AMIs: "ubuntu", "nginx"  
EOT  
}
```

- ✓ Clean, readable, and gives devs a much better DX than cryptic key errors.
-

⚠ Validation Block Limitations

Terraform variable validations **CANNOT** reference resources like this:

```
hcl
CopyEdit
# ❌ This is INVALID:
validation {
  condition = alltrue([
    for config in values(var.ec2_instance_config_map) :
      config.subnet_index < length(aws_subnet.main)
  ])
  error_message = "Invalid subnet index"
}
```

Why?

- Variable validation happens **before** Terraform evaluates the resource graph.
 - Resources (like `aws_subnet.main`) don't exist yet during validation.
-



Terraform's Design Philosophy Here

“Validation blocks are for **purely static checks** on user-provided variable values — not on runtime infrastructure state.”



But You Still Want to Validate Subnet Index?

You'll need:

- A **custom validation function** in a module (complex)
- Or do it at runtime using `locals + dynamic blocks + terraform plan` fail logic (advanced)

This will be covered later in **object validation**.



Clean-Up After Yourself

After validating everything and testing error messages:

```
bash
CopyEdit
terraform destroy
```

Why?

- Avoid EC2 cost
 - Free up subnet allocations
 - Stay cloud-responsible
-

Terraform Function Summary

Function	Purpose
<code>alltrue()</code>	Checks if all booleans in a list are true
<code>contains()</code>	Checks if a list contains a value
<code>values()</code>	Returns just the values of a map

Comparison Table: List vs Map Validation

Aspect	<code>list(object)</code>	<code>map(object)</code>
Syntax	<code>for config in var.list</code>	<code>for config in values(var.map)</code>
Validation Flex	High	High
Resource Matching	<code>count.index</code>	<code>each.key, each.value</code>
Error Localization	Indexed error	Named key helps identify better



Best Practices

- Always validate **user input** at the variable level
 - Use `values(var.map)` when validating `map(object)`
 - Prefer **clear error messages** over cryptic Terraform exceptions
 - Never reference resources in `validation {}` blocks
-

Final Thought

“Fail fast, fail clearly. Terraform validations are your first line of defense in building secure and maintainable infrastructure modules.”



Lecture 9: Refactoring Subnets and EC2 to Use Named Configuration Maps

✳️ What Changed

Old Setup	New Setup
Subnets created via <code>count</code>	Subnets created via <code>for_each</code> on a <code>map</code>
Referenced via index	Referenced via name-based keys (<code>"default"</code> , <code>"subnet1"</code>)
EC2 subnet selection via index	EC2 subnet selection via key in config (<code>subnet_name</code>)

✅ Step-by-Step Improvements

1. 🔥 Remove Obsolete Variables

hcl
CopyEdit

```
# Removed or defaulted
variable "ec2_instance_count"      # gone
variable "ec2_instance_config_list" # now empty default
variable "subnet_count"           # gone
```

2. 🧠 Define `subnet_config` as a Map of Objects

h
CopyEdit

```
variable "subnet_config" {
  type = map(object({
    cidr_block = string
  }))
}
```

⌚ Enables subnets to be created by name: `"default"`, `"subnet1"`, etc.

3. Refactor `aws_subnet` to Use `for_each`

hcl
CopyEdit

```
resource "aws_subnet" "main" {
  for_each = var.subnet_config

  vpc_id          = aws_vpc.main.id
  cidr_block      = each.value.cidr_block
  availability_zone = data.aws_availability_zones.available.names[0]
  tags = {
    Name = each.key
  }
}
```

4. Define `ec2_instance_config_map` to Include Subnet Name

hcl
CopyEdit

```
variable "ec2_instance_config_map" {
  type = map(object({
    ami        = string
    instance_type = string
    subnet_name  = optional(string, "default")
  }))
}
```

5. Refactor EC2 Instance Resource to Use Named Subnet

hcl
CopyEdit

```
resource "aws_instance" "from_map" {
  for_each = var.ec2_instance_config_map

  ami          = local.ami_id[each.value.ami]
  instance_type = each.value.instance_type
  subnet_id     =
  aws_subnet.main[each.value.subnet_name].id
  associate_public_ip_address = true
  tags = {
    Name = each.key
  }
}
```

```
    }
}
```

Example `terraform.tfvars`

```
hcl
CopyEdit
subnet_config = {
  default = {
    cidr_block = "10.0.0.0/24"
  },
  subnet1 = {
    cidr_block = "10.0.1.0/24"
  }
}

ec2_instance_config_map = {
  ubuntu_one = {
    ami           = "ubuntu"
    instance_type = "t2.micro"
    # defaults to subnet "default"
  },
  nginx_one = {
    ami           = "nginx"
    instance_type = "t2.micro"
    subnet_name   = "subnet1"
  }
}
```

Terraform Validations

Validate Subnet CIDR Block Format

Terraform doesn't have `is_valid_cidr()`, so we abuse `cidrnetmask()` with `can()`:

```
hcl
CopyEdit
validation {
  condition = alltrue([

```

```

    for config in values(var.subnet_config) :
        can(cidrnetmask(config.cidr_block))
    )
    error_message = "At least one of the provided CIDR blocks is
invalid."
}

```

 `can()` checks if expression runs *without error*. If `cidrnetmask()` fails, it's invalid.

Map vs Index Referencing — Why This Matters

Feature	Index-based	Map-based (New Way)
Referencing	By position (e.g., <code>count.index</code>)	By name (e.g., <code>subnet_name</code>)
Stability on reordering	 Breaks	 Stable
Code readability	 Confusing	 Clear
Validation support	 Hard to validate	 Easier
Cloud best practice	 Anti-pattern	 Follows IaC design principles

Bonus: Clean Code & Destroy Resources

- Use `terraform destroy` to free AWS resources
- Clean up old `count`-based logic entirely

bash
CopyEdit
`terraform destroy`

Terraform Function Recap

Function	Purpose
----------	---------

<code>can()</code>	Returns <code>true</code> if expression runs without error
<code>cidrnetmask()</code>	Converts CIDR to subnet mask — fails if CIDR is invalid
<code>for_each</code>	Iterate over <code>map</code> resources
<code>optional()</code>	Define optional object fields with default values
<code>each.key/value</code>	Used in resource when iterating via <code>for_each</code>

🎯 Final Thoughts

Terraform is most powerful when **you let the data drive the structure**.

Replace indexes with meaningful names, hardcoded strings with config maps, and unstructured lists with objects.

You've now:

- ✓ Modularized subnet creation
- ✓ Created **stable, reusable EC2 config**
- ✓ Implemented advanced **input validation**
- ✓ Reduced cognitive load by eliminating index-based infrastructure

Project-IAM user management

IAM User Management Project — Lecture 1 Notes

Project Goal

Automate creation of **IAM users**, **roles**, and **role-to-user bindings** using Terraform, driven by a simple **YAML file**.

Core Architecture Breakdown

Input: `user_data.yaml`

```
yaml
CopyEdit
# Example structure
users:
  - name: alice
    roles: [developer, s3_readonly]

  - name: bob
    roles: [admin]
```

- A declarative input format: username and list of roles they can assume.
 - Users = human or service identities.
 - Roles = IAM roles defined elsewhere and linked to permissions.
-

Terraform Design Overview

IAM Roles Block

- Create IAM roles (including trust policy)

- Attach policies (managed or custom)
- Limit role assumption to listed users only !

IAM Users Block

- Create IAM users
- Assign console login credentials (with a password)
- Attach a login profile (optional)
- DO NOT output plaintext passwords in real infra

Role Assumption Mapping

- Based on `user_data.yaml`, define which users can **assume which roles**
- Enforced via **trust policies** in the roles



Implementation Strategy

Step	Component	Description
1	<code>user_data.yaml</code>	Define users and their assigned roles
2	<code>data "yamldecode"</code>	Load and parse the YAML into Terraform
3	<code>aws_iam_user</code>	Create IAM users dynamically using <code>for_each</code>
4	<code>aws_iam_role</code>	Create IAM roles and trust policies
5	<code>aws_iam_user_login_profile</code>	Create passwords (only for testing purposes here)
6	<code>aws_iam_policy_attachment</code>	Attach AWS managed or custom policies
7	Trust Policy for Roles	Only allow assigned users to assume their roles



Key Security Concepts Covered

Concept	Description
IAM Users vs Roles	Users = login identities; Roles = assumed identities with permissions
Assume Role	A user needs explicit trust to assume a role
Trust Policy	JSON document that defines who can assume a role
Password Handling	NEVER output plaintext passwords in production (this is a local-only lab case)

Hints + Tips

Use AWS Managed Policies (Quick Start)

```
hcl
CopyEdit
aws_iam_role_policy_attachment {
  role      = aws_iam_role.example.name
  policy_arn = "arn:aws:iam::aws:policy/ReadOnlyAccess"
}
```

Or Write Custom Inline Policies for Extra Practice

```
hcl
CopyEdit
resource "aws_iam_role_policy" "custom_policy" {
  name    = "custom"
  role    = aws_iam_role.example.id

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [...]
  })
}
```

Real-World Caveats (Important)

Thing

Best Practice

Plaintext Passwords	✗ Avoid in real infra. Use secrets manager or secure channels
Over-Permissive Roles	✗ Never allow wildcard ("*") unless truly needed
Open Trust Policies	✗ Limit Principal in trust policy to specific user ARNs
Hardcoded Role Assumptions	✗ Dynamically generate trusted ARNs from YAML user-role mapping

Terraform Techniques Involved

- `yamldecode(file("user_data.yaml"))`
 - `for_each` with nested maps
 - Dynamic creation of:
 - `aws_iam_user`
 - `aws_iam_role`
 - `aws_iam_role_policy_attachment`
 - `aws_iam_user_login_profile`
 - `aws_iam_policy`
 - Creating trust relationships (policy documents)
 - Using `jsonencode()` for inline IAM policy blocks
-

Summary Workflow

```
yaml  
CopyEdit  
YAML (users + roles)  
↓  
yamldecode in Terraform  
↓
```

```
Create IAM Users (aws_iam_user)
  ↓
Create IAM Roles (aws_iam_role)
  ↓
Attach policies (managed or custom)
  ↓
Generate trust policies
  ↓
Allow users to assume roles they are assigned
```



Final Notes

This is **not just a Terraform exercise** — it's a **mini IAM platform** automation system.

- YAML drives everything → change YAML → rerun Terraform → IAM updated ✓
- Perfect for **multi-user, multi-role** organizations.
- Can be extended later with:
 - MFA
 - SSH key uploads
 - Federated identities
 - SSO/SAML integrations

IAM User Management – Lecture 2 Notes

 Focus: Parse a YAML input file into Terraform using `yamldecode`, and understand why roles > groups

Project Structure Setup

```
sql
CopyEdit
project-02-iam-users/
├── provider.tf
└── users.tf
└── user-roles.yaml
```



user-roles.yaml – Core Input Definition

```
yaml
CopyEdit
users:
  - username: john
    roles: [read_only, developer]

  - username: jane
    roles: [admin, auditor]

  - username: laura
    roles: [read_only]
```

Purpose of This File:

- Stores **human-readable** IAM user + role mapping.
- Allows **non-technical users** (e.g. product owners, managers) to contribute to infra setup **without touching Terraform files**.
- A step toward **configuration-as-data** (infra logic decoupled from infra config).

Why Use Roles Instead of Groups?

Topic	Groups	Roles (Preferred)
Permission	Union of all group policies	Only permissions of assumed role
Flexibility	Always active	Must be explicitly assumed
Security	Dangerous if over-permissive	Much safer by design
Real-world	Good for defaults	Ideal for least-privilege ops

Example:

If John is in `AdminGroup` and clicks "delete instance" by mistake — **it's executed instantly.**

If John **assumes the `read_only` role**, then he can't perform destructive actions — even by mistake.

 **Roles enforce intent. Groups don't.**

Add Schema Reference (Non-functional)

Helps future collaborators understand the structure without digging through code.

```
yaml
CopyEdit
# Brief schema reference:
# users: [
#   {
#     username: string,
#     roles: [read_only | developer | admin | auditor]
#   }
# ]
```

 **Purely documentation**, not used by Terraform directly.

provider.tf

h
CopyEdit

```
terraform {  
    required_version = ">= 1.7.0"  
}
```

- Sets required Terraform version.
 - No AWS provider yet – this lecture is focused purely on YAML parsing and outputs.
-

Parsing YAML in Terraform – Core Technique

♦ users.tf

hcl
CopyEdit

```
locals {  
    users_from_yaml =  
yamldecode(file("${path.module}/user-roles.yaml"))  
}  
  
output "users" {  
    value = local.users_from_yaml.users  
}
```

What Happens:

1. `file()` reads the raw contents of `user-roles.yaml`
 2. `yamldecode()` turns it into native HCL structures (map, list, strings)
 3. `local.users_from_yaml.users` gives the list of user-role objects
-

Output from `terraform plan`

Looks like this:

```
hcl
CopyEdit
users = [
  {
    "roles" = ["read_only", "developer"]
    "username" = "john"
  },
  {
    "roles" = ["admin", "auditor"]
    "username" = "jane"
  },
  {
    "roles" = ["read_only"]
    "username" = "laura"
  }
]
```

- ✓ Human-readable
 - ✓ Easily iterable in Terraform (`for_each`)
 - ✓ Decoupled config logic
-

⚠ Terraform Command Recap

```
bash
CopyEdit
terraform init      # Initialize
terraform fmt       # Auto-format HCL files
terraform plan      # See parsed output
```

🔑 Key Terraform Functions Used

Function	Description
<code>file()</code>	Reads contents of a file
<code>yamldecode()</code>	Converts YAML into HCL native data structures

`local` block For defining derived variables

Why This Lecture Is Crucial

- This is the **data ingestion layer** — everything in the IAM system flows from this YAML input.
 - Clean separation of logic and data = easier scalability, testing, collaboration.
-

Summary of What We Did

Step	Task
1	Created <code>user-roles.yaml</code> with user-role mapping
2	Added schema comment to guide future edits
3	Parsed YAML into Terraform using <code>file()</code> + <code>yamldecode()</code>
4	Output the parsed list to confirm it's working
5	Discussed why roles are more secure than groups

Next Up (Spoiler)

- Create IAM roles (with trust policies)
- Create IAM users
- Allow only assigned users to assume specific roles
- Attach managed or custom policies to roles

IAM User Management – Lecture 3

Notes

 Focus: Parse user info from YAML → Create IAM users in AWS using Terraform

Objective:

Create IAM users dynamically from a YAML file (`user-roles.yaml`) using:

- `aws_iam_user`
 - `for_each` loop based on decoded YAML
-



Input File Reminder – `user-roles.yaml`

```
yaml
CopyEdit
users:
  - username: john
    roles: [read_only, developer]

  - username: jane
    roles: [admin, auditor]

  - username: laura
    roles: [read_only]
```



Step-by-Step Implementation in Terraform

◆ 1. Create IAM users from parsed YAML

```
hcl
CopyEdit
resource "aws_iam_user" "users" {
  for_each = toset([
```

```

        for user in local.users_from_yaml.users : user.username
    ])

    name = each.value
}

```

 **Explanation:**

- `local.users_from_yaml.users` is a list of user objects.
 - `for user in ... : user.username` extracts just the usernames.
 - `toset([...])` is **mandatory** because `for_each` only works with:
 - `map`
 - `set of strings`
 - `each.value` = current username
-

 **2. Provider Fixes**

In `provider.tf`:

```

hcl
CopyEdit
terraform {
  required_version = ">= 1.7.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1" # or your preferred region
}

```

Terraform Commands

```
bash
CopyEdit
terraform init      # install AWS provider
terraform plan      # check creation of IAM users
terraform apply      # create users on AWS
```

Result After Apply

What Terraform creates:

- 3 IAM users: `john`, `jane`, and `laura`
 - Visible in AWS Console → IAM → Users
-

What's Missing?

- Users **cannot log in** yet.
 - No passwords or access credentials are set.
 - This is resolved in the **next lecture** with `aws_iam_user_login_profile`.
-

Terraform Gotchas in This Lecture

Issue	Fix
<code>for_each</code> error (tuple type)	Use <code>toset()</code> to convert list to set
Missing provider block	Add <code>required_providers</code> and <code>provider "aws"</code>

No region configured	Set <code>region = "eu-west-1"</code> (or any valid one)
----------------------	--

Key Terraform Functions Used

Function	Description
<code>for</code> loop	Extracts usernames from YAML-decoded list
<code>toset()</code>	Converts list to set (required for <code>for_each</code>)
<code>each.val</code>	Represents current username in the loop

Why This Pattern Is Powerful

- External config (`YAML`) drives infrastructure creation.
 - Very **declarative** – you change the YAML, run `apply`, and Terraform handles the delta.
 - Enables **non-devs** to manage IAM via config files without touching HCL.
-

Summary

Task	Status
Parse users from YAML	✓
Extract usernames	✓
Create IAM users in AWS	✓
Add provider config	✓
Set up login profile	 Next lecture

Next Up:

- Create IAM login profiles for users.
- Securely generate and output (temporarily) passwords.
- Start attaching roles/policies.

Project: IAM Users – Lecture 4

Objective:

- Generate passwords for IAM users
 - Allow login to AWS Console
 - Output passwords temporarily (insecure in real life but useful for dev/test)
-

Step-by-Step Terraform Implementation

1. Resource: `aws_iam_user_login_profile`

hcl
CopyEdit

```
resource "aws_iam_user_login_profile" "users" {
  for_each = aws_iam_user.users

  user = each.value.name
  password_length = 8

  lifecycle {
    ignore_changes = [
      password_length,
      password_reset_required,
      pgp_key,
    ]
  }
}
```

Explanation:

- `for_each = aws_iam_user.users`: creates a login profile for each IAM user created earlier.
- `password_length = 8`: minimum required password length.

- `lifecycle.ignore_changes`: ensures **login profiles aren't destroyed** if optional fields change later (e.g., password length or reset flags).
-

📝 2. Outputting Passwords (⚠️ Sensitive Data)

```
hcl
CopyEdit
output "passwords" {
  sensitive = true
  value = {
    for user, user_login in aws_iam_user_login_profile.users :
      user => user_login.password
  }
}
```

🧠 Important!

- `sensitive = true`: Required by Terraform 1.3+ to avoid unsafe exposure of secrets.
 - **Don't use in real-world environments** (passwords should be sent via secure email or secret manager).
-

✍️ 3. Running It

```
bash
CopyEdit
terraform plan
terraform apply
```

➡️ Output will say:

```
css
CopyEdit
Changes to Outputs:
  + passwords = (sensitive value)
```

📘 To reveal sensitive output after apply:

```
bash
CopyEdit
terraform output passwords
```

Live AWS Test

To verify:

1. Copy one user password (e.g., for `jane`)
2. Go to AWS Console → Sign out → Sign in as IAM user
3. Use correct **Account ID** and **IAM Username (e.g., jane)**
4. Paste password → Login

 Result:

- Login works 
- But **access is denied** to everything 

Why?

No permissions are granted yet → the user must **assume a role** to gain privileges (covered in next lecture)

Key Concepts Recap

Concept	Purpose
<code>aws_iam_user_login_profile</code>	Enables IAM user console login
<code>password_length</code>	Minimum 8 chars for valid password
<code>lifecycle.ignore_changes</code>	Prevents Terraform from recreating user if defaults change
<code>sensitive = true</code>	Avoids unsafe display of passwords in Terraform output

`terraform output`
passwords

Manual reveal of sensitive values



Real-World Reminder

🚫 DO NOT

Output plaintext passwords

Share login details openly

Hardcode password length
forever

✅ DO INSTEAD

Use PGP encryption or Secrets
Manager

Deliver via secure email/OTP systems

Use variable for flexibility



Summary of Achievements in This Lecture

Task	Status
Created IAM user login profiles	✓
Generated console login passwords	✓
Output passwords securely	✓
Logged into AWS Console as user	✓
Verified zero permissions	✓



Next Up

- Create **IAM roles**
- Assign **trust policies** to allow users to **assume roles**
- Attach **permissions** to roles instead of users

This is where the **magic of role-based access control (RBAC)** begins 🔥



Project: IAM Users – Lecture 5



Objective:

- Understand **AWS Managed Policies**
 - Prepare to **attach permissions** to roles via these policies
 - Explore the **IAM Policy Console** to choose appropriate ARNs
-



Why This Lecture Matters

In the next steps, we'll:

- Create IAM Roles (users will **assume these**)
- Attach **policies** to the roles (to define what actions the role allows)

So, before attaching, we must understand the **types of policies available**.



IAM Permissions Architecture Refresher

Component	Description
User	Can log in but has no permissions by default
Role	Defines a set of permissions
Policy	Attached to a role or user/group to define what they can do
Trust Policy	Controls who/what can assume a role



Where to Explore Policies

AWS Console → IAM → Policies

You'll see:

Type	Description
AWS Managed Policies	Pre-built by AWS for common tasks
Job Function Policies	Bundled permissions for specific job roles (e.g. DataScientist, Billing)
Customer Managed Policies	Custom policies you write yourself

Examples of AWS Managed Policies

Name	Description
ReadOnlyAccess	Grants read-only permissions to most AWS services
AmazonEC2FullAccess	Full permissions for EC2
AdministratorAccess	 Full admin permissions (should be used with caution)
Billing	Grants access to billing console
AmazonS3ReadOnlyAccess	Read-only access to S3 buckets

Case Study: **ReadOnlyAccess**

Why it's perfect:

- Useful for junior users, interns, auditors
- Safe – no destructive permissions
- Covers 2000+ actions like:
 - **Get***
 - **List***
 - **Describe***

How to use:

1. Go to **IAM → Policies**
2. Search for: **ReadOnlyAccess**

Click it → On right pane, find the **ARN**:

cpp
CopyEdit
arn:aws:iam::aws:policy/ReadOnlyAccess

3.

You'll **copy this ARN** to attach to IAM roles via Terraform later.

🎓 Key Takeaways

Concept	Insight
IAM Role	Needs a policy to do anything
IAM User	Gets permissions only by assuming a role
Managed Policy	Saves time, pre-written, secure and maintained by AWS
ReadOnlyAccess	Safest starting point for users needing visibility
ARN	Every policy (even AWS-managed) has a unique ARN you'll reference in Terraform

🧠 Real-World Strategy Tip

Role Type	Recommended Policy
Developer	PowerUserAccess (almost full access without IAM)
Support/Audit	ReadOnlyAccess
Admin	AdministratorAccess
Billing Manager	AWSBillingReadOnlyAccess

Summary of Achievements in This Lecture

Task	Status
Explored AWS IAM policies in console	✓
Understood Managed vs Job Function policies	✓
Found and reviewed ReadOnlyAccess	✓
Learned how to get policy ARNs for Terraform	✓

Next Up

In the next lecture, you'll:

- Create IAM roles using Terraform
- Attach AWS managed policies using the ARNs you explored here
- Set up **trust policies** to allow **specific users** to assume those roles (from YAML config)

Get ready to **tie roles, trust, and policies together** in a powerful access model. 

Lecture 6: Creating IAM Roles

Objective:

- Define **IAM roles** for each user role from the YAML file (`read-only`, `admin`, `developer`, `auditor`)
 - Add dummy **assume role policies**
 - Prepare for next step: attach policies to these roles
-

Step-by-Step Breakdown

1. Define Role → Policy Mappings in `locals`

Use a `locals` block to map role names to relevant AWS-managed policy ARNs:

```
hcl
CopyEdit
locals {
  role_policies = {
    read-only = ["arn:aws:iam::aws:policy/ReadOnlyAccess"]
    admin     = ["arn:aws:iam::aws:policy/AdministratorAccess"]
    auditor   = ["arn:aws:iam::aws:policy/SecurityAudit"]
    developer = [
      "arn:aws:iam::aws:policy/AmazonVPCFullAccess",
      "arn:aws:iam::aws:policy/AmazonEC2FullAccess",
      "arn:aws:iam::aws:policy/AmazonRDSFullAccess"
    ]
  }
}
```

This keeps your Terraform code DRY and allows dynamic role creation.

2. Create IAM Roles Dynamically

Using:

```
hcl
CopyEdit
resource "aws_iam_role" "roles" {
  for_each = toset(keys(local.role_policies))

  name          = each.key
  assume_role_policy =
  data.aws_iam_policy_document.assume_role_policy.json
}
```

This creates **4 roles**: `read-only`, `admin`, `auditor`, and `developer`.

✓ 3. Create Placeholder Trust Policy

Roles **must** have an `assume_role_policy`, even if it allows nobody (yet). Here's the trick:

```
hcl
CopyEdit
data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRole"]

    principals {
      type      = "AWS"
      identifiers = [] # nobody can assume it... yet
    }
  }
}
```

 Reminder: This is a **temporary block** – we'll replace `[]` with real ARNs (i.e., users) in future lectures.

🛠 4. Debugging + Fixes

Issue	Fix
Terraform plan hangs	Caused by invalid trust policy (no principal)
Error: <code>malformed policy document</code>	Add at least 1 valid principal in <code>identifiers</code>

Workaround	Hardcoded Laura's ARN temporarily to test role creation
------------	---

5. Manual Trust Test with One User

To test role assumption:

- Hardcoded **Laura**'s ARN as the trust principal:

```
h
CopyEdit
principals {
    type = "AWS"
    identifiers = ["arn:aws:iam::<account_id>:user/Laura"]
}
```

- Applied and verified:
 - Role was created 
 - Trust relationship was present 

6. Attempted Role Assumption in Console

Steps:

1. Login as **Laura** using password from previous output
2. No permissions initially (as expected)
3. Tried to **Switch Role** to **read-only**
4. **Switch worked** – UI showed role switch
5. **But still access denied**

Why?

Because **no policy was attached to the role** yet.

 A role **without any policy = useless**, even if assumed

Key Takeaways

Concept	Explanation
<code>assume_role_policy</code>	A trust policy defines who can assume a role
Terraform dynamic <code>for_each</code>	Efficiently creates multiple roles based on your config
AWS Managed Policies	Used here to avoid writing custom policies
Temporary Testing	Hardcoded principal for <code>Laura</code> used to test early
Access Denied after AssumeRole	Because roles need actual policies attached!

What's Next

In **Lecture 7**, we will:

- Attach the **correct managed policies** to each IAM role
- Fix the **trust policy dynamically** using the YAML file
- Finalize the connection from YAML → Terraform → AWS

Lecture 7: Attaching IAM Policies to Roles (Terraform)

Goal:

Attach appropriate AWS-managed policies to each IAM role using Terraform — in a way that's **dynamic**, **scalable**, and **clean**.

Step-by-Step Breakdown

1. Problem with Current Format

We had:

```
hcl
CopyEdit
locals {
  role_policies = {
    read-only = [ "ReadOnlyAccess" ]
    admin     = [ "AdministratorAccess" ]
    ...
  }
}
```

 **Problem:** Can't loop easily over `map<string, list<string>>`

 **Solution:** Flatten into a list of role-policy objects.

2. Transform Role-Policy Map into List of Objects

```
hcl
CopyEdit
locals {
  role_policies_list = flatten([
    for role, policies in local.role_policies : [
      for policy in policies : {
        role   = role
        policy = policy
      }
    ]
  ])
}
```

```
    ]
}
```

💡 Now `role_policies_list` looks like:

```
hcl
CopyEdit
[
  { role = "read-only", policy = "ReadOnlyAccess" },
  { role = "admin",     policy = "AdministratorAccess" },
  ...
]
```

This makes it **ideal for iteration** using `count`.

✓ 3. Fetch Policy ARNs Dynamically (Using `data` blocks)

```
hcl
CopyEdit
data "aws_iam_policy" "managed_policies" {
  for_each = toset([for item in local.role_policies_list :
item.policy])
  name      = each.value
}
```

- `toset([...])` removes **duplicate policies**
 - `data.aws_iam_policy.managed_policies["ReadOnlyAccess"].arn` gives you the ARN
-

✓ 4. Attach Policies to Roles Dynamically

Use `aws_iam_role_policy_attachment` with a `count`:

```
h
CopyEdit
resource "aws_iam_role_policy_attachment" "role_policy_attachments"
{
```

```
count      = length(local.role_policies_list)

role       =
aws_iam_role.roles[local.role_policies_list[count.index].role].name
policy_arn =
data.aws_iam_policy.managed_policies[local.role_policies_list[count.
index].policy].arn
}
```

Each element in the list results in **one attachment**.

Terraform Plan Results

Output:

- 6 resources added (1 per role-policy pair)
 - Developer → 3 policies
 - Admin → 1
 - Auditor → 1
 - Read-only → 1

 Permissions granted to roles as per expected mappings.

Demo Recap: Role Assumption Now Works!

- User **Lauro** assumes **read-only** role
- Able to:
 - List EC2/S3 (ReadOnlyAccess)
 -  Cannot create S3 bucket (expected)

 Switching back to user **Lauro**:

- ✗ No access (since user has no permissions)
-

⚠️ But Here's the BIG Problem

🔒 Security Flaw

Right now, *any user* (well, `Lauro`, hardcoded) can assume **any** role (even `admin`)!

🔴 **Bad:** `Lauro` can switch to `admin` role and take over the account.

🔒❓ What's Next?

In upcoming lectures:

- Make **assume role trust policies dynamic** (based on YAML)
 - Ensure only **correct users** can assume **specific roles**
-



Summary Table

Concept	Terraform Block	Notes
Role → Policy Map	<code>locals</code>	Easy to manage manually
Role + Policy Pair List	<code>flatten</code> logic	Needed for looping
Fetch ARNs	<code>data "aws_iam_policy"</code>	Dynamically resolves policy ARNs
Attach Policies	<code>aws_iam_role_policy_attach</code>	Count-based iteration
Validate Access	AWS Console testing	Read-only succeeded, writes blocked
Security Risk	Hardcoded trust (any user → any role)	Will be fixed later

Mental Model

-  Permissions come from Roles
 -  Roles can't be used unless someone is allowed to assume them
 -  We assign permissions to roles (done in this lecture)
 -  We'll now control who can assume which role
-

Let me know if you'd like:

- A full **visual map of role flow**
- A **code export** of the final working state
- A **real-world tip** to prevent role abuse in teams

Ready for Lecture 8? 😎

Ask ChatGPT

Lecture 8: Dynamic Trust Policies – Restrict Who Can Assume Roles

Goal:

Prevent any user from assuming **any role**. Only allow **authorized users** to assume the **specific roles** listed for them in the `users.yaml`.



The Big Problem (from Lecture 7)

- ! All users (e.g., `Lauro`) can assume **all roles**, including `admin`.

This is because the trust policy (assume role policy) allowed a wildcard or hardcoded Arn without filtering based on the user's actual role assignment.

Strategy Breakdown

What We Need:

1. Create a **separate trust policy** for each role.
 2. In that policy, **only list the users** who are allowed to assume that specific role.
-

Part 1: Create One Trust Policy Per Role

hcl

CopyEdit

```
data "aws_iam_policy_document" "assume_role_policies" {
  for_each = toset(keys(local.role_policies)) # ["read-only",
  "admin", "auditor", "developer"]

  statement {
    actions = [ "sts:AssumeRole" ]

    principals {
      type      = "AWS"
      identifiers = [
        local.role_policies[each.key]
      ]
    }
  }
}
```

```
for username in keys(aws_iam_user.users) :  
  
    "arn:aws:iam::${data.aws_caller_identity.current.account_id}:user/${  
        username}"  
        if contains(local.users_map[username], each.key) # ← Only  
if this user is allowed this role  
    ]  
}  
}  
}
```



Part 2: Create a `users_map` for Easy Lookup

Instead of dealing with a list of users like:

```
yaml  
CopyEdit  
- username: john  
  roles: [developer]
```

We convert it to a map like:

```
h  
CopyEdit  
{  
  "john" = [ "developer" ],  
  "laura" = [ "read-only" ],  
  ...  
}
```

Terraform code:

```
hcl  
CopyEdit  
locals {  
  users_map = {  
    for user in local.users_from_yaml :  
      user.username => user.roles  
  }  
}
```

Now, `local.users_map["john"]` gives `["developer"]`



Connect Role to Correct Trust Policy

IAM role definition now uses a **per-role trust policy**:

```
hcl
CopyEdit
resource "aws_iam_role" "roles" {
  for_each          = toset(keys(local.role_policies))
  name              = each.key
  assume_role_policy =
  data.aws_iam_policy_document.assume_role_policies[each.key].json
}
```

This means:

- `read-only` role uses a trust policy that only includes users with the "`read-only`" role.
 - `admin` role includes only "`jane`" if she has the `admin` role, and so on.
-



Supporting Infra

AWS Account ID Dynamically Retrieved

```
hcl
CopyEdit
data "aws_caller_identity" "current" {}
```

Used for creating ARNs dynamically:

```
hcl
CopyEdit
"arn:aws:iam::${data.aws_caller_identity.current.account_id}:user/${username}"
```



Verification via Terraform Plan

✓ What You Should See:

- Read-only role → `laura, john`
 - Developer role → `john`
 - Admin role → `jane`
 - Auditor role → `jane`
 - No users can assume roles not assigned to them
-



Security Example

✗ Before:

```
h  
CopyEdit  
principals {  
    identifiers = [] # everyone (if not filtered)  
}
```

✓ After:

```
hcl  
CopyEdit  
principals {  
    identifiers = [  
        "arn:aws:iam::....:user/john", # if john has that role  
    ]  
}
```



Opinionated Design Decision

Instead of flattening duplicated usernames in YAML:

```
yaml
CopyEdit
- username: john
  roles: [read-only]
- username: john
  roles: [developer]
```

We **disallow duplicates** and raise an error (cleaner + safer):

```
yaml
CopyEdit
- username: john
  roles: [read-only, developer]
```

This:

- Makes YAML easier to review
 - Prevents scattered permissions
 - Easier to revoke access (only one line per user)
-

✓ Final Result

- ✓ Every role has its own trust policy
 - ✓ Only **assigned users** can assume their roles
 - ✓ No hardcoded user ARNs
 - ✓ Dynamic account ID
 - ✓ Secure, scalable, and clean implementation
-

🎯 Summary Table

Feature	Implemented?	Notes
Per-role trust policies	✓	Created via <code>for_each</code>
Limited access per user	✓	<code>contains()</code> check

Dynamic user Arn building	✓	Uses <code>aws_caller_identity</code>
User map for quick access	✓	Transformed from YAML
Duplicate usernames error	✓	Safer than flattening
Final IAM role definition	✓	Fully dynamic

🧠 Mental Model

mermaid

CopyEdit

graph TD

```
A["users.yaml"] -->|parsed| B["users_from_yaml"]
B -->|mapped| C["users_map: username -> [roles]"]
C --> D["for_each IAM policy doc"]
D --> E["role trust policy: only assigned users"]
E --> F["IAM Role (assume_role_policy)"]
F --> G["User can only assume roles assigned"]
```



Lecture 9: Final Testing, Security Deep Dive & Clean Destruction



Full Project Recap



Input:

A simple `users.yaml` file with entries like:

```
yaml
CopyEdit
users:
  - username: john
    roles: [read-only, developer]
  - username: jane
    roles: [admin, auditor]
  - username: laura
    roles: [read-only]
```



System Design Overview

Phase	Terraform Action
 Load	<code>yamldecode()</code> reads <code>users.yaml</code>
 Users	<code>aws_iam_user</code> created per user
 Login	<code>aws_iam_user_login_profile</code> with random password
 Roles	Defined (admin, read-only, etc.)
 Policies	Mapped AWS Managed Policies attached to roles
 Attach	<code>aws_iam_role_policy_attachment</code> per (role, policy)
 Trust Policy	Custom <code>assume_role_policy</code> per role
 Filtering	Only users assigned that role are trusted



```
users_map = { "john" = [ "read-only",  
    "developer" ], ... }
```



Advanced Terraform Concepts Used

Concept	Usage
for_each	Dynamically generate resources
locals	Store transformed data (<code>users_map</code> , <code>role_policies_list</code>)
dynamic blocks	For looping within policy docs
yamldecode()	Load structured user input
contains()	Conditional filtering for trust
data "aws_iam_policy_document" "	Build per-role trust policies
data "aws_iam_policy"	Fetch AWS managed policies
data "aws_caller_identity"	Dynamically get AWS Account ID



Trust Relationship Highlights

For each `IAM Role`, the **trust policy** now:

- Lists **only users** allowed to assume that role (based on `users_map`)
- Example:

```
json  
CopyEdit  
"Principal": {  
    "AWS": [  
        "arn:aws:iam::123456789012:user/jane"  
    ]  
}
```

Testing in AWS Console

User: Laura

- Can **only** assume **read-only**
- **✗** Cannot assume **admin, developer, auditor**

User: Jane

- Can assume **admin** and **auditor**
- **✗** Cannot assume **read-only**

Observation:

Users initially have **no permissions**

- After switching to an allowed role, their permissions are activated
 - This ensures **least privilege principle** is enforced
-

Security Deep Dive

Principle	How it's Applied
 Least Privilege	Users can't assume unassigned roles
 Short-lived creds	Roles use session-based tokens
 No plaintext secrets	(Passwords should never be output in real world!)
 One-line teardown	<code>terraform destroy</code> fully removes access
 No hardcoded ARNs	ARNs built dynamically using data sources

Bonus Insight:

Role assumption uses temporary credentials (token-based), meaning even if compromised, the blast radius is limited to a few hours. 

⭐ The Power of `terraform destroy`

- Destroys:
 - All IAM users
 - All roles
 - All attachments
 - All trust policies
- 🔒 After destroy:
 - Logins to Jane/Laura/John fail (users don't exist anymore)
 - Switch Role = Bad Request

Result:

Single command, total teardown. Zero trace left.

📊 Architecture Summary (Mental Model)

mermaid

CopyEdit

flowchart TD

```
A["users.yaml"] --> B["users_from_yaml"]
B --> C["users_map (username => [roles])"]
C -->|used in| D["trust policies"]
D -->|allow| E["user assumes role"]
E -->|inherits| F["AWS Managed Policies"]
F -->|defined in| G["role_policies list"]
```

🏁 Final Thoughts

This project taught:

Practical Terraform Skills

- Iterations, maps, transformations, conditionals

Real-World IAM Design

- Secure, minimal-permission role-based access
- Clean role-to-user mapping logic
- Dynamic and extensible user definition structure

End-to-End Infra Control

- From YAML input → to IAM users → to controlled access → to complete destroy
-

Terraform Tips from Lecture 9

Tip	Why It Matters
Use maps over lists for lookup	Easier + cleaner logic
Throw errors for dupes	Better security and clarity
Always test assumptions in console	Confirms end-to-end correctness
Never output passwords	Use secure distribution mechanisms
Prefer temporary credentials	Boosts security with expiration

Congratulations!

You now have:

-  Deep Terraform knowledge
-  Secure, dynamic IAM setup
-  Real-world AWS practice
-  End-to-end automation mastery

Terraform modules

Lecture 1: Introduction to Terraform Modules

1. What are Modules?

- **Definition:**

Modules in Terraform are **collections of Terraform configuration files** (usually `.tf` files) stored together in a directory.

They bundle multiple resources that are commonly used together.

- **Root Module vs Child Modules:**

- **Root Module:** The set of `.tf` files in your main working directory (i.e., the current Terraform project).
- **Child Modules:** Additional modules called by the root module. They can be stored **locally** (on your machine) or **remotely** (e.g., Terraform Registry, Git repos).

- **Note:**

Everything you have done so far with Terraform — writing `.tf` files and running Terraform commands — has been working within the **root module**.

2. Why Use Modules?

Modules help **organize, encapsulate, and reuse** Terraform code effectively.

Key Reasons:

- **Organization:**

Modules allow you to **group related infrastructure resources** together logically. This makes your Terraform codebase easier to understand and maintain.

- **Encapsulation:**

- Modules **hide internal implementation details** and **prevent unintended changes** by others.

- They provide a **clear interface** through well-defined **inputs (variables)** and **outputs**.
 - This reduces mistakes especially in complex infrastructure projects.
- **Reusability:**
Modules enable you to **reuse sets of resources** across projects or environments, saving time and improving consistency.
 - **Best Practices Enforcement:**
 - Complex configurations and resource wiring can be **abstracted and standardized** within modules.
 - Modules can **codify best practices** and ensure that these are consistently followed when others use the module.
 - **Publication and Sharing:**
Once created, modules can be:
 - Published **publicly** (e.g., Terraform Registry) for community use.
 - Shared **privately** in internal registries for team or organizational reuse.
-

3. Summary Points

Concept	Explanation
Module	Collection of <code>.tf</code> files grouped as one logical unit.
Root Module	The main module containing the top-level Terraform configuration.
Child Module	Modules called from the root module, local or remote.
Organization	Grouping related resources for clarity and maintainability.
Encapsulation	Hiding complex details and exposing only needed inputs/outputs.
Reusability	Using the same module multiple times to avoid duplication and errors.
Best Practices	Ensuring consistent standards by embedding them in module code.
Publication	Sharing modules publicly or privately for collaboration and reuse.

4. Important Concepts and Terminology

- **Inputs:**
Variables defined in a module to accept configuration parameters.
 - **Outputs:**
Values that a module returns after deployment to expose important info.
 - **Interface:**
The combined inputs and outputs that the users interact with when using a module.
-

5. Additional Notes

- Even a single `.tf` file or set of `.tf` files in a directory is technically a module (the root module).
 - Modules help **reduce human error** by restricting access to internal resources and standardizing configurations.
 - The more complex the infrastructure, the more beneficial using modules becomes.
-

6. Code Example

No specific code was provided in this lecture, but here is a simple example of calling a child module:

```
h
CopyEdit
module "vpc" {
  source = "./modules/vpc"  # Local child module path

  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
}
```

- **Explanation:**
 - `module "vpc"` declares a module call named `vpc`.

- `source` points to the local folder `./modules/vpc` where the child module code resides.
- Variables like `cidr_block` and `enable_dns_support` are inputs to the module.

Lecture 2: Standard Module Structure in Terraform

1. Overview: Why Module Structure Matters

- The **standard module structure** is a **recommended minimal set of files** that Terraform modules should have.
 - This structure is **not mandatory** but is a best practice suggested by HashiCorp/Terraform to ensure:
 - Consistency
 - Ease of understanding
 - Better integration with tools like Terraform Registry
 - You can add more files as needed, but the minimal recommended files form a solid foundation.
-

2. Recommended Minimal Set of Files in a Module

File Name	Purpose & Details
LICENSE	Specifies usage rights and licensing terms for the module.
main.tf	The main entry point where the bulk of resources are defined.
outputs.tf	Contains all module outputs (values returned after apply).
variables.tf	Contains all variables exposed by the module for input params.
README.md	Documentation describing the module and usage examples.

3. Detailed Explanation of Each File

LICENSE File

- Explains **how others can use your module** legally.

- Important if you plan to **publish modules publicly** or share within teams.
 - Common licenses include MIT, Apache 2.0, etc.
-

main.tf File

- Acts as the **primary Terraform configuration file** within the module.
 - If your module is **simple**, all resources can be placed here.
 - For **complex modules**, you may split resources into multiple files named for clarity, e.g.:
 - vpc.tf for VPC resources
 - subnet.tf for subnet resources
 - The filename `main.tf` is a **convention** rather than a requirement.
 - You may use meaningful file names to organize resources better.
-

outputs.tf File

- Contains **output blocks** for the module.
 - These outputs expose information back to the root module or to other modules.
 - Important for passing data like resource IDs, IPs, ARNs, etc.
 - Though the filename is not enforced by Terraform, naming it `outputs.tf` helps Terraform Registry **auto-generate documentation**.
-

variables.tf File

- Contains **variable declarations** that specify input parameters for the module.
- Allows users to customize module behavior via input variables.

- Naming it `variables.tf` enables the Terraform Registry to auto-generate docs.
-

README.md File

- Provides **detailed documentation** about the module.
 - Should describe:
 - What the module does
 - Required and optional variables
 - Outputs
 - Example usage snippets
 - Terraform Registry uses this file to show module info clearly.
-

4. Additional Notes

- This structure supports **modularity, clarity, and reusability**.
 - Proper documentation and file organization help with **team collaboration and ease of maintenance**.
 - Terraform Registry benefits from this structure by:
 - Displaying inputs and outputs nicely
 - Showing usage examples
 - Enabling users to understand modules without digging into code
-

5. Example Minimal Module Structure

```
arduino
CopyEdit
/my-module
├── LICENSE
└── main.tf
```

```
|__ variables.tf  
|__ outputs.tf  
└__ README.md
```

Lecture 3: Using Public Modules in Terraform Projects

1. Introduction to Terraform Registry

- **Terraform Registry** is the central public repository where you can **browse and discover Terraform modules**.
 - Modules can be **publicly available** or hosted in **private registries** (e.g., via Terraform Cloud) for internal/team use.
 - Both public and private registries work similarly — they host modules with documentation and versioning.
-

2. Browsing the Terraform Registry

- On the main page, you can toggle between:
 - **Providers** — plugins that manage resources of specific platforms (AWS, Azure, GCP, etc.)
 - **Modules** — reusable collections of Terraform resources.
 - **Modules depend on Providers** — a module specifies the providers it needs, and your Terraform config must declare those providers as well.
-

3. What Are Modules on the Registry?

- Modules are **wrappers around provider resources**, bundling multiple resources to deliver specific infrastructure features.
- They provide:
 - A **clear interface** (inputs & outputs)
 - **Encapsulation** of complex wiring

- Better developer experience
-

4. Example: Terraform AWS VPC Module

- Popular public module on Terraform Registry for creating VPCs on AWS.
 - Each module has:
 - **Versions** (e.g., 5.5.3) to ensure **stability** and prevent breaking changes.
 - **Documentation** covering inputs, outputs, and example usage.
-

5. Understanding Module Inputs and Outputs

- **Inputs = Variables exposed by the module**
 - Can have **default values** or be **required** (no default).
 - Can be marked **sensitive** to protect secrets.
 - Have descriptions to explain their purpose.
 - Example: `cidr_block` variable for the VPC module.
 - If no input is provided, default `10.0.0.0/16` is used.
 - **Outputs = Values exposed by the module**
 - These are returned after module deployment.
 - Must have a value and can have descriptions.
 - Auto-generated documentation helps users understand available outputs.
-

6. Module Documentation on the Registry

- Auto-generated from the module's:

- `variables.tf` (inputs)
 - `outputs.tf` (outputs)
- Includes examples on how to use the module.
 - Shows dependencies:
 - **External dependencies:** modules outside the current repo.
 - **Internal (submodules):** modules inside the same repo, not considered external.
-

7. Provider Dependencies

- The module declares which **provider versions** it requires.
 - Your root Terraform configuration must include compatible provider versions.
-

8. Practical Usage Summary

- To use a module from the registry, declare a `module` block in your Terraform configuration:

```
hcl
CopyEdit
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.3"

  cidr = "10.0.0.0/16"
  # Other inputs here...
}
```

- You only need to specify **required inputs or override defaults**.
- Outputs from the module can be accessed as `module.vpc.<output_name>`.

9. How to Learn and Troubleshoot Modules

- You don't need to memorize all inputs/outputs — check the registry docs per module.
 - Use community resources like **StackOverflow**, **GitHub issues**, and **blogs** for help.
 - Good modules have extensive docs and example usage.
-

10. Key Takeaways

Concept	Explanation
Terraform Registry	Central place to find public/private Terraform modules.
Modules vs Providers	Modules wrap resources from providers; providers manage actual resources.
Versioning	Modules have versions to avoid breaking changes in your configs.
Inputs (Variables)	Configuration parameters; can be required or have defaults.
Outputs	Values exposed by modules for use elsewhere.
Documentation	Auto-generated from module files; critical for understanding module usage.
Dependencies	Modules may depend on other modules or providers.

Lecture 4: Deep Dive into Source Code of a Public Terraform Module (AWS VPC)

1. Overview

- This lecture focuses on exploring the **source code of a public Terraform module**, specifically the **Terraform AWS VPC module**.
 - The source code is hosted on **GitHub**, linked from the Terraform Registry.
 - Understanding the structure and contents of the module helps you:
 - Learn module best practices.
 - Customize or extend modules.
 - Use modules effectively.
-

2. Module Repository Structure

- The repository has a **simple and clean structure**, typical for Terraform modules.
 - Key files present at the root level:
 - `main.tf` — primary file where resources are declared.
 - `variables.tf` — contains all module variables (inputs).
 - `outputs.tf` — declares outputs returned by the module.
 - `LICENSE` — licensing information.
 - `README.md` — documentation (same content as the Terraform Registry page).
 - These files **follow the minimal module structure best practice** discussed earlier.
 - No deep folder nesting for resources, simplifying navigation.
-

3. main.tf File

- Contains the **majority of resource declarations**.
 - Organized with **comments grouping resources** by their purpose (e.g., subnets, routing tables).
 - Contains **local variables (locals)**:
 - Some are **common locals** defined near the top.
 - Others are **resource-specific locals**, scoped closer to their resources.
 - Use of locals helps **simplify complex expressions** and **reduce repetition**.
 - Comments and structure are mainly for **developer readability** — do not affect Terraform usage.
-

4. variables.tf File

- Lists **all variables declared** by the module.
 - Variables have:
 - **Descriptions** which appear automatically on the Terraform Registry inputs documentation.
 - Default values, if provided, making some variables optional.
 - Some variables are **required** (no default), making them mandatory for users.
 - Maintaining detailed and clear descriptions here is **important for user understanding**.
-

5. outputs.tf File

- Contains **all outputs exposed by the module**.
- Outputs provide valuable info such as resource IDs, ARNs, or attributes.

- These outputs are **auto-documented** on the registry.
 - Users can reference outputs in their root Terraform code using `module.<name>.<output>`.
-

6. `versions.tf` File

- Specifies **required provider versions** for the module (e.g., AWS provider versions).
 - Does **not specify module versioning** (that's handled externally via registry and Git tags).
-

7. `CHANGELOG.md` File

- Records **module version history**:
 - Lists changes, additions, fixes per version.
 - Helpful to track evolution and impact of upgrades.
-

8. Examples Folder

- Contains **usage examples** illustrating how to consume the module.
 - Includes:
 - **complete example** — a full-featured, complex configuration.
 - **simple example** — minimal setup for quick understanding.
-

8.1 Simple Example Breakdown

- Demonstrates creating a **small VPC** with:
 - `cidr_block`

- A subset of availability zones (AZs)
- Subnets

Demonstrates **slicing a list of availability zones** to select a subset dynamically:

hcl

CopyEdit

```
slice(data.aws_availability_zones.available.names, 0, 3)
```

-
- Explanation of `slice` function:
 - Extracts elements from a list between `start_index` (inclusive) and `end_index` (exclusive).
 - This allows region-compatibility (e.g., selecting 3 AZs out of possibly 6).
- Passing of **tags** to resources is shown to propagate metadata consistently.
- No input variables declared specifically in the simple example — uses defaults or hardcoded values.

9. Exploring Outputs & Variables in Examples

- Outputs give detailed info about created resources even in simple setups.
- Variables for examples are often minimal or none, focusing on straightforward usage.
- Versions file reiterates required providers for compatibility.

10. Submodules

- The AWS VPC module contains **submodules**, e.g. `vpc_endpoints`.
- Submodules are **modules nested inside the main module repository** under the `modules/` directory.
- Submodules have:

- Their own `main.tf`, `variables.tf`, `outputs.tf`, `README.md`.
- Do **not** have separate license or changelog (inherit from root).
- Submodules can be used **independently**, e.g. just creating VPC endpoints without the whole VPC.
- They provide **modular, smaller functionality** inside a bigger module.

In Terraform source declarations, submodules can be accessed with:

hcl

CopyEdit

```
source = "terraform-aws-modules/vpc/aws//modules/vpc_endpoints"
```

- - The double slash `//` indicates navigating inside the module repo folder structure.
-

11. Summary & Best Practices

- Module source code exploration is crucial to:
 - Understand **how modules work internally**.
 - Learn **good code structuring and modularity**.
 - Inspect **inputs/outputs** and their documentation.
 - See real-world usage of **locals and resource grouping**.
 - Maintaining **well-documented variables and outputs** improves user experience.
 - Using **submodules** helps break down complex infrastructure into manageable parts.
 - Examples are invaluable to help users quickly understand module usage.
 - Versioning and changelogs help maintain stability and track changes over time.
-

12. Next Steps

- Next lecture will cover **practical hands-on usage** of public modules like the AWS VPC module.
 - You are encouraged to:
 - Explore GitHub repos of modules you plan to use.
 - Read the examples and documentation.
 - Experiment with simple and complete examples locally.
-

If you want, I can also prepare a step-by-step guide for set

Lecture 5: Practical Use of the Public AWS VPC Module

1. Setup: Create a Working Directory and Files

Create a new directory to organize your project:

```
bash
CopyEdit
mkdir "12_public_modules"
cd "12_public_modules"
```

•

Inside this directory, create a Terraform configuration file:

```
bash
CopyEdit
touch networking.tf
```

•

- Open `networking.tf` to start coding the module usage.

2. Using the AWS VPC Module

Define a module block in `networking.tf` to use the AWS VPC module from the Terraform registry:

```
h
CopyEdit
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.3"
}
```

•

- Note:

- The module name (`vpc`) can be anything; it is an identifier local to your config.

- The source points to the Terraform registry path for the AWS VPC module.
 - Pin the version to ensure consistency with examples (5.5.3 here).
-

3. Initialize Terraform and Download Modules

Run:

```
bash
CopyEdit
terraform init
```

- What happens:
 - Terraform downloads the module source code into `.terraform/modules`.
 - You will see the version being downloaded.
 - Terraform downloads any required providers (e.g., AWS provider).
 - You can inspect the downloaded files (e.g., `variables.tf`, `outputs.tf`) as they mirror the GitHub repo.
-

4. Provider Version Compatibility Issues

- If you specify a provider version constraint in your root Terraform project that conflicts with the module's required provider version, `terraform init` will error out.

Example of incompatible constraint:

```
hcl
CopyEdit
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "5.0.0"    # pinned exactly to 5.0.0
    }
}
```

```
}
```

-

Module requires AWS provider >= 5.20.0, so pinning to exactly 5.0.0 causes:

```
javascript
CopyEdit
Error: Incompatible provider version
```

- Terraform cannot have two different versions of the same provider simultaneously.
 - **Resolution:** Make sure root project provider version constraints satisfy module requirements.
-

5. Fix Provider Version Constraint

Modify provider constraints in root configuration:

```
hcl
CopyEdit
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.20"  # satisfy module's minimum version
    }
  }
}
```

-

Run:

```
bash
CopyEdit
terraform init -upgrade
```

- Terraform will now successfully initialize using compatible provider versions.
-

6. Formatting Terraform Code

Run:

```
bash
CopyEdit
terraform fmt
```

- - This cleans up code formatting in your .tf files for better readability.
-

7. Configuring the VPC Module Inputs

Pass necessary variables to the module to customize the VPC:

```
hcl
CopyEdit
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.3"

  cidr = "10.0.0.0/16"

  private_subnets = ["10.0.1.0/24"]
  public_subnets  = ["10.0.128.0/24"]

  tags = {
    Name = "12-public-modules"
  }
}
```

- - Notes on subnet CIDR choices:
 - Dividing the 10.0.0.0/16 block roughly in half:
 - Private subnets: 10.0.0.0 to 10.0.0.127 (for example)
 - Public subnets: 10.0.128.0 to 10.0.255.255
 - This is a **design choice**, not a Terraform or AWS requirement.
-

8. Adding the AWS Provider Block

In a file like provider.tf, configure AWS provider and region:

```
hcl
CopyEdit
provider "aws" {
  region = "eu-west-1"
}
```

- Make sure your AWS credentials are properly set (e.g., via environment variables or `~/.aws/credentials`).
-

9. Dealing with Availability Zones (AZs)

Error encountered during plan:

```
pgsql
CopyEdit
Error: Index 0 out of range for list
```

- This happens because the module expects a list of availability zones in the variable `availability_zones` (or similar), but none were provided.

Fix by querying available AZs dynamically:

```
hcl
CopyEdit
data "aws_availability_zones" "available" {
  state = "available"
}
```

-

Pass the AZ names to the module:

```
hcl
CopyEdit
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.3"
```

```
cidr          = "10.0.0.0/16"
private_subnets = ["10.0.1.0/24"]
public_subnets  = ["10.0.128.0/24"]
availability_zones = data.aws_availability_zones.available.names

tags = {
  Name = "12-public-modules"
}
}
```

•

10. Running Terraform Plan

Execute:

```
bash
CopyEdit
terraform plan
```

-
- Terraform shows all the resources that will be created, including:
 - VPC resource
 - Public and private subnets
 - Route tables and associations
 - Internet gateway
 - Security groups
 - Default route tables
 - The module encapsulates many AWS resources **under the hood**, saving you from writing them manually.
-

11. Benefits & Recommendations

- The module provides **abstraction and encapsulation**:

- You don't need to manually configure each AWS resource for a VPC.
 - Greatly simplifies management of complex infrastructure.
 - It's **recommended to understand VPC fundamentals** before using modules:
 - Inspect what the module creates.
 - Know AWS VPC concepts to troubleshoot or customize further.
 - Modules improve **developer productivity** and enforce **best practices** by encapsulating complexity.
-

12. Summary

- Created a working directory and Terraform files.
- Defined and sourced the AWS VPC public module with version pinning.
- Handled provider version constraints and initialization errors.
- Configured module variables: CIDR blocks, subnets, availability zones.
- Used a data source to fetch AZs dynamically.
- Ran plan successfully and reviewed resources created by the module.
- Understood the power of modules for abstraction and reuse.

Lecture 6: God-Level Detailed Notes with Full Code Explanation — Deploying EC2 Instance Using AWS Public Modules and VPC Integration

1. Objective of the Lecture

- Deploy an **EC2 instance** in AWS using the **official Terraform AWS EC2 module**.
 - Use the **VPC module** from **Lecture 5** to get networking info (subnets, security groups).
 - Refactor code to use **locals** for shared constants (CIDR blocks, project name, tags).
 - Demonstrate how to **chain modules** by passing outputs from one module as inputs to another.
 - Understand **Terraform data sources** (for AMIs).
 - Learn best practices for Terraform module use and management.
-

2. Project Structure and Files

- We now have 3 important files:
 - `shared_locals.tf` — store shared constants (project name, CIDR blocks, tags).
 - `networking.tf` — contains VPC module usage.
 - `compute.tf` — contains EC2 instance module usage.
-

3. Refactoring Hardcoded Values to Locals

Why?

- Hardcoding values inside modules repeatedly is bad.
 - Locals provide a **single source of truth** for constants.
 - Makes updates easy and consistent.
-

Code: `shared_locals.tf`

```
hcl
CopyEdit
locals {
  project_name = "12_public_modules"

  vpc_cidr = "10.0.0.0/16"

  private_subnet_cidrs = ["10.0.1.0/24"]
  public_subnet_cidrs  = ["10.0.128.0/24"]

  common_tags = {
    Project    = local.project_name
    ManagedBy = "Terraform"
  }

  instance_type = "t2.micro"
}
```

4. Using VPC Module with Locals

- Use `local.vpc_cidr`, `local.private_subnet_cidrs`, `local.public_subnet_cidrs`.
- Pass common tags from `local.common_tags`.

Code snippet: `networking.tf`

```
hcl
CopyEdit
```

```

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.3"

  name = local.project_name

  cidr          = local.vpc_cidr
  private_subnets = local.private_subnet_cidrs
  public_subnets  = local.public_subnet_cidrs

  tags = local.common_tags
}

```

5. Getting the Latest Ubuntu AMI Using Data Source

Why?

- AMI IDs change between regions and over time.
- Hardcoding AMI IDs causes issues.
- Using a **data source** allows fetching the most recent Ubuntu AMI dynamically.

Code snippet:

```

hcl
CopyEdit
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name    = "name"
    values =
    ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  filter {
    name    = "virtualization-type"
    values = ["hvm"]
  }

```

```
    owners = ["099720109477"] # Canonical's AWS Account ID
}
```

6. Deploy EC2 Module: Detailed Code & Explanation

- Source: public Terraform AWS EC2 instance module.
- Version pinned to **5.6.1** (stable during lecture).
- **Inputs explained:**

Input Name	Value & Explanation
name	Local project name for resource naming
ami	AMI ID obtained from data source above
instance_type	From <code>local.instance_type</code> ("t2.micro")
vpc_security_group_ids	List containing default security group from VPC module outputs (allows SSH, etc.)
subnet_id	First public subnet ID from VPC module outputs
tags	Common tags from locals

Code snippet: `compute.tf`

```
hcl
CopyEdit
module "ec2" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "5.6.1"

  name        = local.project_name
  ami         = data.aws_ami.ubuntu.id
  instance_type = local.instance_type

  # Security groups must be a list, so wrap in square brackets
  vpc_security_group_ids = [module.vpc.default_security_group_id]

  # Deploy in first public subnet from VPC module outputs
```

```
    subnet_id = module.vpc.public_subnets[0]

    tags = local.common_tags
}
```

7. How Terraform Modules Communicate: Using Outputs

- The VPC module exposes useful outputs, for example:

```
hcl
CopyEdit
output "default_security_group_id" {
  description = "ID of the default security group."
  value        = aws_security_group.default.id
}
```

- Access these outputs in other modules with:

```
hcl
CopyEdit
module.vpc.default_security_group_id
module.vpc.public_subnets
```

- This is how we **chain modules** together.
-

8. Commands to Run

- After adding a new module, **always run**:

```
bash
CopyEdit
terraform init
```

- This downloads the EC2 module.
- Then check plan:

```
bash
CopyEdit
terraform plan
```

- Verify changes: EC2 instance creation and VPC resources.
- Finally, apply changes:

```
bash
CopyEdit
terraform apply
```

- Confirm with `yes`.
-

9. Verifying in AWS Console

- Go to EC2 dashboard:
 - One running instance should appear.
 - Instance should be associated with the deployed VPC and subnet.
 - VPC dashboard:
 - Verify subnets and route tables.
 - Public and private subnets have explicit route table associations.
-

10. Destroying Infrastructure

- When done, clean resources:

```
bash
CopyEdit
terraform destroy
```

- Confirm with `yes`.
 - Note: Default Network ACLs managed by AWS are removed from state but not deleted.
-

11. Summary of Concepts Learned

Concept	Description
Terraform Locals	Centralized, reusable constants to avoid duplication.
Terraform Modules	Encapsulated infrastructure components for reuse.
Terraform Data Sources	Dynamically fetch info like latest AMIs.
Module Outputs	Share important resource info between modules.
Module Version Pinning	Use specific versions for stable, reproducible builds.
Resource Association	Subnets linked to route tables explicitly for clarity.
Terraform Lifecycle Commands	<code>init</code> , <code>plan</code> , <code>apply</code> , <code>destroy</code> for managing infrastructure.

12. Full Example Combined Code

```
hcl
CopyEdit
# shared_locals.tf
locals {
    project_name = "12_public_modules"

    vpc_cidr = "10.0.0.0/16"

    private_subnet_cidrs = ["10.0.1.0/24"]
    public_subnet_cidrs  = ["10.0.128.0/24"]

    common_tags = {
```

```
    Project      = local.project_name
    ManagedBy   = "Terraform"
}

instance_type = "t2.micro"
}

# networking.tf
module "vpc" {
    source  = "terraform-aws-modules/vpc/aws"
    version = "5.5.3"

    name          = local.project_name
    cidr         = local.vpc_cidr
    private_subnets = local.private_subnet_cidrs
    public_subnets  = local.public_subnet_cidrs

    tags = local.common_tags
}

# data source for AMI
data "aws_ami" "ubuntu" {
    most_recent = true

    filter {
        name    = "name"
        values =
        ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
    }

    filter {
        name    = "virtualization-type"
        values = ["hvm"]
    }

    owners = [ "099720109477" ] # Canonical
}

# compute.tf
module "ec2" {
    source  = "terraform-aws-modules/ec2-instance/aws"
```

```
version = "5.6.1"

name          = local.project_name
ami           = data.aws_ami.ubuntu.id
instance_type = local.instance_type

vpc_security_group_ids = [module.vpc.default_security_group_id]
subnet_id            = module.vpc.public_subnets[0]

tags = local.common_tags
}
```

Lecture 7: Terraform Modules — Best Practices & Deep Dive

1. What is a Terraform Module?

- A module is simply a folder with `.tf` files defining resources, variables, outputs.
 - But **not all modules are well-designed** just because they exist.
 - Modules are abstractions to group related resources for reuse, clarity, and encapsulation.
-

2. When to Create a Module?

Key signals to create a module:

- **Logical Abstraction:** When a set of resources logically belong together as a reusable building block.
Example: VPC + subnets + routing = network module.
 - **Strongly Coupled Resources:** Resources that always get created together and depend on each other should be grouped.
 - **Encapsulation Needs:** Hide complex internal details from users, exposing only necessary configs for simplicity.
-

3. Why Build Good Modules?

- Improves **developer experience** — easy to use and maintain.
- Avoids **repetition** and **hardcoding** across Terraform projects.
- Supports **reusability** across multiple environments or teams.

- Facilitates **collaboration** by hiding complexity but exposing stable interfaces.
-

4. Best Practices When Building Terraform Modules

4.1 Use Object Type Variables for Related Inputs

Instead of many separate variables, group related variables into **objects**.

```
hcl
CopyEdit
variable "db_config" {
  type = object({
    username = string
    password = string
    port     = number
  })
}
```

- Benefits:
 - Clear structure and domain separation.
 - Easier to extend and maintain.
 - Less cluttered input variable list.

4.2 Separate Long-Lived vs Short-Lived Infrastructure

- Keep **stable** resources (like S3 buckets, database instances) in one module.
 - Keep **frequently changing** resources (like EC2 instances) separate.
 - This reduces unnecessary resource replacements when unrelated parts change.
-

4.3 Do Not Try to Cover Every Edge Case

- Modules should be **general enough for reuse**, but not cluttered with every niche option.
 - Catering to every edge case leads to **complex, hard-to-maintain modules**.
 - Instead, keep modules **focused** and handle edge cases via composition or wrapper modules.
-

4.4 Limit the Number of Configurable Variables

- Don't expose every internal detail as a variable.
 - Only expose variables that need to be configurable by users.
 - This maintains **encapsulation** and reduces user confusion.
-

4.5 Output as Much Useful Information as Possible

- Even if you don't immediately use some output, provide it.
 - Especially for **public modules** where users might need unexpected info.
 - It helps debugging, integration, and extension.
-

4.6 Define a Stable Input and Output Interface

- Variables and outputs form the module's API.
 - Changing variable or output names breaks downstream users.
 - Design **carefully and consistently** to minimize breaking changes.
 - Version your modules to manage evolution safely.
-

4.7 Document Variables and Outputs Thoroughly

- Every variable should have a **description** explaining its purpose.
 - Outputs should also be documented.
 - Good docs help users understand and use your modules faster and correctly.
-

4.8 Prefer a Flat and Composable Module Structure

- Avoid deeply nested modules (modules calling modules calling modules).
 - Instead:
 - Keep modules flat and focused.
 - Compose infrastructure by combining modules at the root level.
 - Easier maintenance, debugging, and scaling.
-

4.9 Make Assumptions Explicit with Validation & Conditions

- Don't blindly trust user inputs.
- Use Terraform's **validation blocks** and **custom conditions** to validate inputs.
- This prevents invalid infrastructure deployment.

Example:

```
hcl
CopyEdit
variable "instance_count" {
  type    = number
  default = 1

  validation {
    condition      = var.instance_count > 0
    error_message = "instance_count must be greater than zero."
  }
}
```

4.10 Make Module Dependencies Explicit via Input Variables

- Avoid hidden or implicit dependencies using `data` sources inside modules.
 - Instead, pass dependencies explicitly via variables.
 - This improves clarity, reusability, and makes the module's contract clear.
-

4.11 Keep Module Scope Narrow and Focused

- Don't try to solve everything with one giant module.
 - Keep modules **small, specific, and composable**.
 - Build complex infrastructure by combining small modules.
-

5. Summary Table of Best Practices

Practice	Why It Matters
Use object type variables	Cleaner inputs, easier to extend
Separate long-lived and short-lived infra	Reduce unintended resource changes
Avoid covering all edge cases	Keeps modules simple and maintainable
Limit exposed variables	Preserve abstraction and reduce confusion
Output useful info extensively	Helps users and integration
Stable input/output interface	Prevent breaking changes
Thorough documentation	Speeds up adoption and reduces errors
Flat module structure	Easier maintenance and understanding
Validate inputs explicitly	Avoids invalid deployments
Explicit dependencies via inputs	Clear contracts and improved reusability
Narrow module scope	Keeps modules focused and composable

6. Final Thoughts

- Designing modules well is **more important** than just knowing Terraform syntax.
- Good modules **save time and headaches** in the long term.
- Practice these best practices consistently for professional Terraform projects.
- Keep learning and reviewing community-recommended practices.

Lecture 8: Building Your Own Local VPC Module in Terraform — Step 1

Overview & Goal

- So far you've used **public Terraform modules** (like the AWS VPC module).
 - Now we start building a **custom local Terraform module** — beginning with a **networking module**.
 - This module will:
 - Create a **VPC with a given CIDR block**.
 - Allow configuring **multiple subnets**.
 - Let users mark subnets as **public or private**.
 - Accept user inputs for **CIDR blocks** and **Availability Zones (AZs)** per subnet.
 - We will build this step-by-step across exercises.
-

Why build your own module?

- Encapsulate complexity (e.g., creating IGW, route tables, associations automatically).
 - Improve usability — user just marks subnet as public or not, no need to manage resources manually.
 - Aligns with best practices from Lecture 7.
 - Enables reusability and easier maintenance.
-

1. Setup Project Folder Structure

- At the root of your project (not inside previous folders like `06-resources`), create:

```
css
CopyEdit
13-local-modules/
└── networking/
    ├── README.md
    ├── LICENSE
    ├── main.tf
    ├── variables.tf
    ├── outputs.tf
    └── providers.tf
```

- This is the **standard module structure** discussed before.
-

2. Writing the README.md for Module

- Describe what your module does:

```
markdown
CopyEdit
# Networking Module
```

This module creates a VPC with a configurable CIDR block.

It supports multiple subnets, where each subnet can be marked as public or private.

Users provide CIDR blocks and availability zones per subnet.

The module manages associated resources such as Internet Gateway, Route Tables, and Route Table Associations automatically for public subnets.

3. Define Providers in providers.tf

```

hcl
CopyEdit
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

```

- This ensures your module knows to use AWS provider version 5.0 or above.
- Define a variable for region in `variables.tf`:

```

hcl
CopyEdit
variable "aws_region" {
  type      = string
  description = "AWS region to deploy resources in"
  default    = "eu-west-1" # example default
}

```

4. Define Variables in `variables.tf`

VPC CIDR block variable with validation:

```

hcl
CopyEdit
variable "vpc_cidr" {
  type      = string
  description = "CIDR block for the VPC"

  validation {
    condition    = can(cidrnetmask(var.vpc_cidr))
  }
}

```

```
        error_message = "The variable vpc_cidr must be a valid CIDR
block."
    }
}
```

- **cidrnetmask** is a Terraform built-in function that returns the subnet mask length of a CIDR, errors if invalid.
 - Using **validation** here **fails early during terraform plan**, avoiding wasted apply attempts and AWS API errors.
-

5. Formatting Terraform Code Recursively

- Run:

```
bash
CopyEdit
terraform fmt -recursive
```

- This formats all **.tf** files including nested modules/directories.
-

6. Test Module Locally by Using it in Root Configuration

Create a root-level **.tf** file (e.g., **main.tf**) like:

```
hcl
CopyEdit
module "networking" {
  source    = "./modules/networking"  # Local path to your module
  folder
  vpc_cidr = "10.0.0.0/16"
  aws_region = "eu-west-1"
}
```

- This imports your local module.
 - Terraform will immediately warn if **required variables are missing**.
-

7. Initializing Terraform & Module Installation

- Run:

```
bash
CopyEdit
terraform init
```

- This downloads any providers and initializes your configuration.
 - Note: For local modules, it just verifies existence but you must still run init to prepare providers.
-

8. Run Terraform Plan

```
bash
CopyEdit
terraform plan
```

- At this early stage, since you haven't created any resources in your module yet, Terraform will show **no changes**.
 - If invalid CIDR is given (e.g. `vpc_cidr = "invalid"`), Terraform will error out due to validation.
-

9. Why Validation is Important

- Without the validation block on `vpc_cidr`, Terraform would try to apply and AWS API would reject the invalid CIDR.

- Validation lets us catch errors **earlier and cleaner**.
 - Avoids wasted time and unnecessary resource creation failures.
-

10. Summary so far:

- You created a local module scaffold with the typical structure.
 - You declared AWS provider requirements.
 - Defined a mandatory variable `vpc_cidr` with validation.
 - Tested module usage in a root Terraform configuration.
 - Learned how to `terraform init` and `terraform plan` on the local module.
 - Validated input variables prevent invalid configurations.
-

What's next?

- In the **next lecture**, you'll start implementing actual Terraform resources inside your module (`main.tf`), like:
 - Create the AWS VPC resource.
 - Add subnet resources.
 - Add internet gateway, route tables, and associations.
 - Implement logic to distinguish between public/private subnets automatically.
-

Bonus: Minimal `main.tf` to start creating a VPC resource inside your module

```
hcl
CopyEdit
resource "aws_vpc" "this" {
  cidr_block = var.vpc_cidr
  tags = {
    Name = "my-vpc"
  }
}
```

- This will create the actual VPC based on the user input CIDR.
- You can run `terraform plan` to confirm Terraform recognizes the resource to be created.

Lecture 9: Migrating to Object Variables in Your Terraform Module

Goal of this Exercise

- Improve the module interface by migrating **multiple related primitive input variables** into a **single object variable**.
 - Learn how to use **object-type variables** in Terraform for better organization, extensibility, and readability.
 - See how this change affects module usage and why it's a good practice.
-

1. Current Setup: Using Primitive Variables

Your module currently has separate variables like:

```
hcl
CopyEdit
variable "vpc_cidr" {
  type = string
  validation {
    condition      = can(cidrnetmask(var.vpc_cidr))
    error_message = "The variable vpc_cidr must be a valid CIDR
block."
  }
}

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

- These variables are passed independently.
- Example usage in `main.tf` or `vpc.tf` inside the module:

hcl

```
CopyEdit
resource "aws_vpc" "this" {
  cidr_block = var.vpc_cidr
  tags = {
    Name = var.vpc_name
  }
}
```

2. Testing Current Module

- Pass variables when calling the module:

```
hcl
CopyEdit
module "networking" {
  source    = "./modules/networking"
  vpc_cidr = "10.0.0.0/16"
  vpc_name = "13-local-modules"
}
```

- Run:

```
bash
CopyEdit
terraform init
terraform apply
```

- Terraform provisions a VPC with the CIDR and name passed.
-

3. Migrating to an Object Variable for VPC Configuration

Why migrate?

- Logical grouping: related attributes belong together.

- Easier to extend without breaking interface.
- Cleaner variable namespace.
- Better alignment with best practices from Lecture 7.

How?

- Define a **single object variable** called `vpc_config`:

```

hcl
CopyEdit
variable "vpc_config" {
  type = object({
    cidr_block = string
    name       = string
  })

  validation {
    condition      = can(cidrnetmask(var.vpc_config.cidr_block))
    error_message = "The cidr_block in vpc_config must be a valid
CIDR block."
  }
}

```

4. Update Usage of the Variable Inside Module

Update resource references:

```

hcl
CopyEdit
resource "aws_vpc" "this" {
  cidr_block = var.vpc_config.cidr_block
  tags = {
    Name = var.vpc_config.name
  }
}

```

Remove the old `vpc_cidr` and `vpc_name` variables since they're now part of the object.

5. Update Module Usage in Root Configuration

Change from:

```
hcl
CopyEdit
module "networking" {
  source    = "./modules/networking"
  vpc_cidr = "10.0.0.0/16"
  vpc_name = "13-local-modules"
}
```

To:

```
hcl
CopyEdit
module "networking" {
  source      = "./modules/networking"
  vpc_config = {
    cidr_block = "10.0.0.0/16"
    name       = "13-local-modules"
  }
}
```

6. Notes on Breaking Changes and Versioning

- Changing from multiple variables to an object is a **breaking interface change**.
- For published modules (e.g., in Terraform Registry), this requires:
 - Incrementing the **major version**.
 - Clients to **update their module calls** to use the new interface.
- But this design is more scalable for the future: you can add new fields inside the object without breaking the interface.

7. Test and Validate

- Run:

```
bash
CopyEdit
terraform plan
```

- You should see **no changes** to your infrastructure since only the interface changed, not the resources.
- Run:

```
bash
CopyEdit
terraform fmt -recursive
```

- Ensures formatting is clean across all files.
-

8. Best Practice Reminder

- Start thinking and organizing your variables in terms of **objects** rather than many unrelated primitives.
 - Group variables **logically by functionality** or domain.
 - This improves maintainability, clarity, and extensibility.
-

9. Summary of Key Code Changes

variables.tf

```
hcl
CopyEdit
variable "vpc_config" {
  type = object({
    cidr_block = string
```

```

    name      = string
  })

validation {
  condition  = can(cidrnetmask(var.vpc_config.cidr_block))
  error_message = "The cidr_block in vpc_config must be a valid
CIDR block."
}
}

```

vpc.tf (or main.tf in module)

```

hcl
CopyEdit
resource "aws_vpc" "this" {
  cidr_block = var.vpc_config.cidr_block
  tags = {
    Name = var.vpc_config.name
  }
}

```

Root module usage

```

hcl
CopyEdit
module "networking" {
  source      = "./modules/networking"
  vpc_config = {
    cidr_block = "10.0.0.0/16"
    name       = "13-local-modules"
  }
}

```

Lecture 10: Extending VPC Module with Subnet Configuration and `for_each` Loop

Introduction

This lecture extends your Terraform VPC module by adding support for **multiple subnets**. The goal is:

- Allow users to provide subnet details in a map format.
 - Use a `for_each` loop to create subnets dynamically.
 - Add validation on CIDR blocks inside the subnet map.
 - Introduce availability zone configuration.
 - Lay groundwork to differentiate public/private subnets in the future.
-

Step 1: Understanding Why a Map of Objects?

- A **single subnet config as an object** only supports one subnet.
 - A **map of objects** allows **multiple named subnets**, e.g., "`subnet_one`", "`subnet_two`".
 - Each subnet has its own config (CIDR, AZ).
 - Map keys are identifiers; values are subnet configs.
-

Step 2: Defining the `subnet_config` Variable (with validation)

Open `variables.tf` inside `modules/networking`, then add:

```

hcl
CopyEdit
variable "subnet_config" {
  type = map(object({
    cidr_block = string
    az         = string # Availability zone (e.g., eu-west-1a)
  }))
}

validation {
  condition = alltrue([
    for config in values(var.subnet_config) :
      can(cidrnetmask(config.cidr_block))
  ])
  error_message = "Each subnet's cidr_block must be a valid CIDR
block."
}

description = "Map of subnet configurations keyed by subnet name"
}

```

Explanation:

- `type` specifies a map where values are objects with two strings: `cidr_block` and `az`.
 - The `validation` block ensures all CIDR blocks are valid using `cidrnetmask()`.
 - `alltrue` requires **all subnet configs** pass the check.
 - This catches errors **before** Terraform applies — better UX.
-

Step 3: Updating Your Module Call (Root Module)

In the root terraform configuration (outside the module), pass subnet config like:

```

hcl
CopyEdit
module "networking" {
  source = "./modules/networking"
}

```

```

vpc_config = {
  cidr_block = "10.0.0.0/16"
  name       = "13-local-modules"
}

subnet_config = {
  subnet_one = {
    cidr_block = "10.0.0.0/24"
    az         = "eu-west-1a"
  }
  subnet_two = {
    cidr_block = "10.0.1.0/24"
    az         = "eu-west-1b"
  }
}
}

```

Explanation:

- Users define multiple subnets inside a map.
 - Each subnet has its CIDR and AZ.
 - Easy to add/remove subnets by modifying this map.
-

Step 4: Creating AWS Subnets Using `for_each`

In the module's Terraform files (e.g., `vpc.tf` or `subnets.tf`), create subnet resources dynamically:

```

hcl
CopyEdit
resource "aws_subnet" "this" {
  for_each = var.subnet_config

  vpc_id      = aws_vpc.this.id
  cidr_block   = each.value.cidr_block
  availability_zone = each.value.az

```

```
tags = {
    Name = "${var.vpc_config.name}-${each.key}"
}
}
```

Explanation:

- `for_each` iterates over each key-value pair of `subnet_config`.
 - `each.key` is the subnet name, e.g., "subnet_one".
 - `each.value` is the object holding subnet properties.
 - Subnet is created with specified CIDR and AZ.
 - Tags include VPC name + subnet key for easier identification.
-

Step 5: Testing Your Configuration

Formatting files

Run in your module root:

```
bash
CopyEdit
terraform fmt -recursive
```

to format all files, including nested modules.

Running Terraform commands

1. Initialize Terraform (if not done already):

```
bash
CopyEdit
terraform init
```

2. Validate plan:

```
bash
CopyEdit
terraform plan
```

- If the `subnet_config` has invalid CIDRs, you get an error from your validation block.
- If valid, you see the plan to create subnets.

Example error when CIDR is invalid:

```
vbnetwork
CopyEdit
Error: Invalid value for variable
|
|   on variables.tf line XX:
|     (variable "subnet_config") Validation failed: Each subnet's
cidr_block must be a valid CIDR block.
```

Step 6: How the Abstraction Benefits the User

- User only defines subnet configurations in one place.
 - No need to understand subnet creation details.
 - The module encapsulates:
 - Linking subnets to VPC ID.
 - Associating AZs.
 - Tagging naming conventions.
 - Extensible to add more features (e.g., public/private flags) without changing interface.
-

Step 7: What's Next?

- Implement subnet **public/private** flags.

- Add **Internet Gateway**, **Route Tables**, and **associations**.
 - Validate availability zones properly.
 - Enhance outputs to expose subnet IDs.
 - Update Readme with subnet usage examples.
-

Full Code Summary

variables.tf in modules/networking

```
hcl
CopyEdit
variable "vpc_config" {
  type = object({
    cidr_block = string
    name       = string
  })

  validation {
    condition = can(cidrnetmask(var.vpc_config.cidr_block))
    error_message = "The vpc_config.cidr_block must be a valid CIDR
block."
  }
}

variable "subnet_config" {
  type = map(object({
    cidr_block = string
    az         = string
  }))
}

validation {
  condition = alltrue([
    for config in values(var.subnet_config) :
      can(cidrnetmask(config.cidr_block))
  ])
}
```

```
    error_message = "Each subnet's cidr_block must be a valid CIDR
block."
}
}
```

vpc.tf or subnets.tf in modules/networking

hcl
CopyEdit

```
resource "aws_vpc" "this" {
  cidr_block = var.vpc_config.cidr_block

  tags = {
    Name = var.vpc_config.name
  }
}

resource "aws_subnet" "this" {
  for_each = var.subnet_config

  vpc_id          = aws_vpc.this.id
  cidr_block      = each.value.cidr_block
  availability_zone = each.value.az

  tags = {
    Name = "${var.vpc_config.name}-${each.key}"
  }
}
```

Usage Example in Root Module

hcl
CopyEdit

```
module "networking" {
  source = "./modules/networking"

  vpc_config = {
    cidr_block = "10.0.0.0/16"
    name       = "13-local-modules"
  }
}
```

```
subnet_config = {
  subnet_one = {
    cidr_block = "10.0.0.0/24"
    az         = "eu-west-1a"
  }
  subnet_two = {
    cidr_block = "10.0.1.0/24"
    az         = "eu-west-1b"
  }
}
```

If you want, I can also provide explanations for each step of the Terraform lifecycle with this setup, or help you write the README with examples. Let me know!

Ask ChatGPT

Lecture 11: Validating Availability Zones in a Terraform VPC Module Using Lifecycle Preconditions

1. Context & Objective

- You have a Terraform module that creates a VPC subnet.
 - The module takes **availability zones (AZs)** as input.
 - AWS requires that these AZs **must be valid** and belong to the region you are working with.
 - The goal is to **validate these AZ inputs early**, during Terraform plan/apply, so invalid AZs don't cause failures from the AWS API.
-

2. Problem Without Validation

- If you provide an invalid AZ (e.g., `eu-central-1b` in the `eu-west-1` region):
 - `terraform plan` does not fail.
 - `terraform apply` fails with AWS error, which is less user-friendly and happens late.

Goal: Catch this **early** and provide **clear validation errors**.

3. Terraform Data Source: `aws_availability_zones`

Terraform provides a data source to fetch all availability zones in the current AWS region:

hcl

CopyEdit

```
data "aws_availability_zones" "available" {  
    state = "available"
```

```
}
```

- This fetches a list of AZs (names like `eu-west-1a`, `eu-west-1b`, etc.) that are currently available in your AWS region.
 - You can access the list with:
`data.aws_availability_zones.available.names`
-

4. Validation with Lifecycle Precondition

Terraform 1.2+ supports lifecycle **precondition** blocks to run validations **before resource creation**.

You can add a precondition to a resource to assert conditions on input variables.

Example: Adding Precondition to Subnet Resource

hcl

CopyEdit

```
resource "aws_subnet" "this" {
  for_each = var.subnet_config

  # existing config here...

  lifecycle {
    precondition {
      condition      =
contains(data.aws_availability_zones.available.names, each.value.az)
      error_message = "Invalid AZ"
    }
  }
}
```

- `contains(list, value)` returns true if the `value` exists in the `list`.
- `each.value.az` is the AZ from your subnet input.

- If the AZ is **not** contained in the list of available AZs, Terraform **stops** before apply, showing `Invalid AZ`.
-

5. Improving the Error Message

Simple error messages like "`Invalid AZ`" aren't very user-friendly.

You can provide a **multi-line, detailed error message** using heredoc syntax (`<<EOT ... EOT`) and **dynamic interpolation** of variables.

Multi-line Friendly Error Message Example:

```
hcl
CopyEdit
lifecycle {
  precondition {
    condition =
contains(data.aws_availability_zones.available.names, each.value.az)

    error_message = <<EOT
The AZ '${each.value.az}' provided for subnet '${each.key}' is
invalid.
The AWS region '${data.aws_availability_zones.available.region}'
supports the following AZs:
[${join(", ", data.aws_availability_zones.available.names)}]
EOT
  }
}
```

Explanation:

- `${each.value.az}` → The invalid AZ input.
- `${each.key}` → The subnet key identifier.
- `${data.aws_availability_zones.available.region}` → Current AWS region.

- `${join(", ", data.aws_availability_zones.available.names)}` → List of valid AZs joined by commas.
 - This produces a clear message explaining which AZ was invalid, for which subnet, and what the valid options are.
-

6. Terraform Function: `join`

- The `join` function concatenates elements of a list into a string with a separator.

Syntax:

```
hcl
CopyEdit
join(separator, list)
```

Example:

```
hcl
CopyEdit
join(", ", ["eu-west-1a", "eu-west-1b", "eu-west-1c"])
```

Produces:

```
arduino
CopyEdit
"eu-west-1a, eu-west-1b, eu-west-1c"
```

This is useful to display lists inside error messages clearly.

7. Practical Workflow

1. Define the `aws_availability_zones` data source.
2. Add the lifecycle precondition in the `aws_subnet` resource.
3. Compose a detailed error message using Terraform expressions and `join`.

4. Run `terraform plan`:
 - If AZ is invalid, plan **fails immediately** with a helpful message.
 - If AZ is valid, plan passes normally.
 5. Run `terraform apply` only when inputs are valid.
-

8. Why Not Use validation Blocks?

- Terraform input `validation` blocks inside variables are limited.
 - They **cannot access data sources** or dynamically compose lists.
 - Lifecycle preconditions **can access data sources** and resource attributes.
 - Thus, preconditions are better for **context-aware, compositional validations**.
-

9. Example Final Resource Block

```
hcl
CopyEdit
resource "aws_subnet" "this" {
  for_each = var.subnet_config

  vpc_id      = var.vpc_id
  cidr_block   = each.value.cidr_block
  availability_zone = each.value.az

  lifecycle {
    precondition {
      condition =
        contains(data.aws_availability_zones.available.names, each.value.az)

      error_message = <<EOT
The AZ '${each.value.az}' provided for subnet '${each.key}' is
invalid.
EOT
    }
  }
}
```

```

The AWS region '${data.aws_availability_zones.available.region}'
supports the following AZs:
[ ${join(", ", data.aws_availability_zones.available.names)} ]
EOT
    }
}
}

```

10. Summary of Key Concepts

Concept	Explanation
aws_availability_zones data source	Fetches the list of AZs available in the current region dynamically.
Lifecycle precondition	Allows validating conditions before resource creation, preventing invalid applies early.
contains() function	Checks if a list contains a given element (used to validate if AZ is valid).
join() function	Joins a list into a string with a separator (used for error message formatting).
User-friendly error messages	Detailed, clear error messages improve user experience and reduce debugging time.
Validation block limitations	Cannot access external data sources or dynamic context, so lifecycle preconditions are preferred.

11. Additional Tips

- Always provide **contextual error messages** with:
 - The invalid value
 - The expected valid options
 - The input key or name
- Use lifecycle preconditions especially when validation depends on **external data or computed values**.

- Test invalid and valid inputs to ensure your module fails safely and clearly.
 - Avoid generic errors — instead aim to help the user **fix the input** quickly.
-

Congratulations!

You've learned how to:

- Use Terraform **data sources** inside modules.
- Implement **lifecycle preconditions** for advanced input validation.
- Compose **dynamic, multi-line error messages**.
- Improve user experience by failing early with clear errors.

This is a powerful pattern for writing **robust, user-friendly Terraform modules**.

Lecture 12: Extending Terraform VPC Module to Support Public & Private Subnets with Internet Gateway and Route Table

1. Objective & Scope

- Extend the Terraform VPC module to support both **public** and **private** subnets.
 - Add logic to **conditionally create an Internet Gateway (IGW)** only if there is at least one public subnet.
 - Associate public subnets with a **public route table** that routes traffic through the IGW.
 - Discuss **module design best practices** for handling optional subnet types.
-

2. Why This Matters

- Public subnets require internet access, so an IGW is essential.
 - Private subnets don't get an IGW.
 - Managing resources conditionally improves:
 - Resource efficiency (no unnecessary IGWs)
 - Module reusability
 - Clear separation of subnet types
 - Allows users to mark any subnet as `public = true` or leave it private by default.
-

3. Step-by-Step Implementation

Step 1: Add `public` Property to Subnet Input Variable

- Modify your subnet input variable type (usually an object or map) to optionally include a `public` boolean.

```
hcl
CopyEdit
variable "subnet_config" {
  type = map(object({
    cidr_block = string
    az        = string
    public     = optional(bool, false) # Optional, defaults to false
  (private subnet)
  }))
}
```

- Users mark a subnet as public by setting `public = true`.
- If omitted, subnet is considered private.

Step 2: Filter Public Subnets Locally

- Create a `local` value to filter only public subnets:

```
hcl
CopyEdit
locals {
  public_subnets = {
    for key, config in var.subnet_config :
      key => config if config.public
  }
}
```

- This produces a map of only those subnets with `public = true`.
- Useful to conditionally create IGW and route tables.

Step 3: Output Public Subnets (Optional Debugging)

- To verify filtering, add output inside module:

```
hcl
CopyEdit
output "public_subnets" {
  value = local.public_subnets
}
```

- Or propagate this output through the root module to view it with `terraform output`.

Step 4: Conditionally Create Internet Gateway

- Create IGW resource with a `count` based on whether there are any public subnets:

```
hcl
CopyEdit
resource "aws_internet_gateway" "this" {
  count = length(local.public_subnets) > 0 ? 1 : 0
  vpc_id = aws_vpc.this.id
}
```

- If `public_subnets` is empty → `count = 0` → no IGW created.
- If at least one public subnet → `count = 1` → one IGW created.
- Prevents multiple IGWs if multiple public subnets exist.

Step 5: Create Public Route Table Conditionally

```
hcl
CopyEdit
resource "aws_route_table" "public" {
```

```

count  = length(local.public_subnets) > 0 ? 1 : 0
vpc_id = aws_vpc.this.id

route {
  cidr_block = "0.0.0.0/0"
  gateway_id = aws_internet_gateway.this[0].id
}
}

```

- Route table directs all outbound traffic (`0.0.0.0/0`) to the IGW.
 - Note indexing `[0]` on IGW resource since it's created with `count`.
-

Step 6: Associate Public Subnets with Public Route Table

- Create route table associations for each public subnet:

```

hcl
CopyEdit
resource "aws_route_table_association" "public" {
  for_each = local.public_subnets

  subnet_id      = aws_subnet.this[each.key].id
  route_table_id = aws_route_table.public[0].id
}

```

- This attaches each public subnet explicitly to the public route table.
 - Private subnets remain implicitly associated with the main route table.
-

4. Handling Terraform `count` Indexed Resources

- When using `count`, access resources via `[index]`, e.g.,

hcl

CopyEdit

```
aws_internet_gateway.this[0].id
```

- Important because `count` can create multiple or zero resources, so direct `.this` is invalid.
-

5. Sample User Configuration

hcl

CopyEdit

```
subnet_config = {
  subnet1 = {
    cidr_block = "10.0.1.0/24"
    az        = "eu-west-1a"
    public     = false
  }
  subnet2 = {
    cidr_block = "10.0.2.0/24"
    az        = "eu-west-1b"
    public     = true
  }
}
```

- Here, `subnet2` is public → triggers IGW creation and public route table.
 - `subnet1` is private → no public routing.
-

6. Terraform Plan & Apply

- Running `terraform plan` shows:
 - IGW resource created only if public subnet exists.
 - Route table and associations created accordingly.
- Running `terraform apply` provisions resources.

- If no public subnets → IGW and public route table **not created**.
-

7. Error Handling & Debugging

- Common mistakes:
 - Forgetting to add `vpc_id` to IGW (causes apply errors).
 - Not indexing IGW or route table when using `count`.
 - Misaligned subnet keys when referencing `aws_subnet.this[each.key].id`.
-

8. AWS Console Verification

- After apply, check AWS Console → VPC → Subnets.
 - Public subnet:
 - Should have **explicit association** to the public route table.
 - Route table should contain route to IGW (`0.0.0.0/0`).
 - Private subnet:
 - Associated implicitly with main route table.
 - No route to IGW.
-

9. Best Practices and Design Considerations

- **Use optional input variables** with sensible defaults to allow backward compatibility.
- **Filter and local values** for clear conditional logic instead of complex `count/for_each` everywhere.

- **Clear separation** of subnet types to maintain modularity and clarity.
 - Always **use descriptive output values** to expose useful info for downstream modules or users.
 - Use **lifecycle preconditions** (from previous lecture) for validating inputs such as AZs alongside this design.
 - Add **comments** in code for readability and maintenance.
-

10. Summary

Step	Key Action
Add subnet <code>public</code> var	Optional bool with default <code>false</code>
Filter public subnets	Use <code>local.public_subnets</code> to extract public ones
Create IGW conditionally	<code>count = length(local.public_subnets) > 0 ? 1 : 0</code>
Create public route table	Conditional route table linked to IGW
Associate subnets	Route table association for all public subnets
Index resources	Use <code>[0]</code> index when accessing IGW and route table with <code>count</code>
Verify & debug	Check plan, apply, AWS console for expected resource setup

11. Full Example Snippet

```

hcl
CopyEdit
variable "subnet_config" {
  type = map(object({
    cidr_block = string
    az        = string
    public     = optional(bool, false)
  }))
}

locals {

```

```

public_subnets = {
    for key, config in var.subnet_config : key => config if
config.public
}
}

resource "aws_internet_gateway" "this" {
count  = length(local.public_subnets) > 0 ? 1 : 0
vpc_id = aws_vpc.this.id
}

resource "aws_route_table" "public" {
count  = length(local.public_subnets) > 0 ? 1 : 0
vpc_id = aws_vpc.this.id

route {
cidr_block = "0.0.0.0/0"
gateway_id = aws_internet_gateway.this[0].id
}
}

resource "aws_route_table_association" "public" {
for_each = local.public_subnets

subnet_id      = aws_subnet.this[each.key].id
route_table_id = aws_route_table.public[0].id
}
}

```

Lecture 13: Creating and Managing Useful Outputs in Terraform VPC Module

1. Lecture Objective

- Improve the VPC module by adding **useful outputs**.
 - Currently, outputs reflect only input subnet config, which is not very helpful.
 - Goal: Expose meaningful, curated information for **VPC ID**, **public subnets**, and **private subnets**.
 - Make outputs user-friendly and easy to consume by downstream modules or root Terraform project.
-

2. Why Outputs Matter

- Outputs allow external Terraform configurations to **reference resources** created inside a module.
- Especially important for networking where many resources depend on subnet IDs, VPC ID, AZs, etc.
- Output design should:
 - Expose essential info (e.g., subnet IDs, availability zones).
 - Avoid overwhelming users with every resource attribute (reduce cognitive load).
 - Use **maps with user-defined keys** to preserve stable referencing.
 - Provide clear **descriptions** for each output.

3. Current Problem

- Existing output only reflects what user passes in as variables.
 - No resource identifiers like subnet IDs or availability zones returned.
 - No way for root project to connect with created AWS subnets or VPC.
-

4. Step-by-Step Output Design

Step 1: Output the VPC ID

- Most important output: the ID of the created VPC.

```
hcl
CopyEdit
output "vpc_id" {
  description = "The AWS ID of the created VPC"
  value        = aws_vpc.this.id
}
```

- This allows referencing the VPC in other resources.
-

Step 2: Output Public Subnets as Map

- Filter **public subnets** from input config using locals (already filtered earlier):

```
hcl
CopyEdit
locals {
  public_subnets = {
    for key, config in var.subnet_config : key => config if
    config.public
  }
}
```

- Create an output that returns a map where:
 - The key is the user-defined subnet key.
 - The value is an object containing:
 - Subnet ID (from created AWS subnet resource).
 - Availability Zone (from AWS resource or input config).

```

hcl
CopyEdit
output "public_subnets" {
  description = "Map of public subnet keys to subnet IDs and
availability zones"
  value = {
    for key in keys(local.public_subnets) : key => {
      subnet_id      = aws_subnet.this[key].id
      availability_zone = aws_subnet.this[key].availability_zone
    }
  }
}

```

- Why a map?
 - Users can reliably reference subnets by the keys they defined.
 - Maintains stable, predictable output order.
 - Easier to use than a list (which may reorder).
-

Step 3: Output Private Subnets as Map

- Similarly, filter private subnets (those where `public` is false or omitted):

```

hcl
CopyEdit
locals {
  private_subnets = {

```

```
        for key, config in var.subnet_config : key => config if
!lookup(config, "public", false)
    }
}
```

- Output private subnets in the same format:

```
hcl
CopyEdit
output "private_subnets" {
  description = "Map of private subnet keys to subnet IDs and
availability zones"
  value = {
    for key in keys(local.private_subnets) : key => {
      subnet_id      = aws_subnet.this[key].id
      availability_zone = aws_subnet.this[key].availability_zone
    }
  }
}
```

5. Why Include Availability Zone?

- Availability zone is critical for:
 - Creating resources colocated with subnets.
 - Understanding subnet distribution.
 - It adds value without overwhelming the user.
-

6. Best Practices for Outputs

- **Descriptions:** Always add descriptions to explain each output.
- **Curate data:** Avoid exposing unnecessary internal resource attributes.
- **Use maps keyed by user input:** Preserves stable references, easier for users.

- **Keep locals close to outputs:** Organizes code logically.
 - **Expose only what's needed:** Balance between transparency and complexity.
-

7. Example Expanded Subnet Configuration

User can define:

```
hcl
CopyEdit
subnet_config = {
  subnet1 = {
    cidr_block = "10.0.1.0/24"
    az         = "eu-west-1a"
    public     = false
  }
  subnet2 = {
    cidr_block = "10.0.2.0/24"
    az         = "eu-west-1b"
    public     = true
  }
  subnet3 = {
    cidr_block = "10.0.3.0/24"
    az         = "eu-west-1c"
    public     = false
  }
  subnet4 = {
    cidr_block = "10.0.4.0/24"
    az         = "eu-west-1a"
    public     = true
  }
}
```

Outputs will then clearly show which subnets are public or private by key, ID, and AZ.

8. Sample Output After `terraform apply`

shell

```

CopyEdit
public_subnets = {
    subnet2 = {
        subnet_id = "subnet-0a12b345cde6789f0"
        availability_zone = "eu-west-1b"
    }
    subnet4 = {
        subnet_id = "subnet-0b23c456def7890a1"
        availability_zone = "eu-west-1a"
    }
}

private_subnets = {
    subnet1 = {
        subnet_id = "subnet-0123abcd4567ef890"
        availability_zone = "eu-west-1a"
    }
    subnet3 = {
        subnet_id = "subnet-0234bcde5678fa901"
        availability_zone = "eu-west-1c"
    }
}

vpc_id = "vpc-0123456789abcdef0"

```

9. Additional Notes

- You can extend outputs with other useful subnet attributes (ARNs, CIDR blocks, etc.) if needed.
 - Always keep user experience in mind: avoid overwhelming users with too much data.
 - For public modules (used by many), more info may be needed.
 - For private/internal modules, you can keep outputs minimal and focused.
-

10. Summary Table

Output Name	Description	Format
vpc_id	ID of the created VPC	string
public_subnets	Map of public subnet keys to subnet IDs & AZs	map(object)
private_subnets	Map of private subnet keys to subnet IDs & AZs	map(object)

Complete Example Snippet (Outputs.tf)

```

hcl
CopyEdit
locals {
  public_subnets = {
    for key, config in var.subnet_config : key => config if
    config.public
  }

  private_subnets = {
    for key, config in var.subnet_config : key => config if
    !lookup(config, "public", false)
  }
}

output "vpc_id" {
  description = "The AWS ID of the created VPC"
  value       = aws_vpc.this.id
}

output "public_subnets" {
  description = "Map of public subnet keys to subnet IDs and
  availability zones"
  value = {
    for key in keys(local.public_subnets) : key => {
      subnet_id      = aws_subnet.this[key].id
      availability_zone = aws_subnet.this[key].availability_zone
    }
  }
}

```

```
output "private_subnets" {
  description = "Map of private subnet keys to subnet IDs and
availability zones"
  value = {
    for key in keys(local.private_subnets) : key => {
      subnet_id      = aws_subnet.this[key].id
      availability_zone = aws_subnet.this[key].availability_zone
    }
  }
}
```

If you want, I can also help you with:

- Refactoring the entire module with these outputs
- Examples of how to consume these outputs in root Terraform code
- Tips for testing and validating outputs

Lecture 14: Wrapping Up Your Terraform Module — License, README, and Examples Folder

1. Purpose of This Lecture

- Finalize your Terraform module by:
 - Adding a **license file**
 - Creating a meaningful **README.md**
 - Organizing an **examples/** folder with practical usage examples
-

2. Why Add a License?

- Defines **legal terms** for usage, reuse, modification, and distribution.
 - Crucial for open-source modules or shared internal projects.
 - Protects **authors** and informs **users** about rights and restrictions.
 - Different licenses have different implications:
 - **MIT License** (used here): Very permissive, allows commercial use, modification, distribution with minimal restrictions.
 - **GPL License**: Requires derivative works to also be open-source (copyleft).
 - **Proprietary licenses**: Restrictive, may forbid reuse or commercial use without permission.
-

How to Add License:

- Add a **LICENSE** file in your module directory.

- Copy the chosen license text (e.g., MIT License).
- Update copyright year and author.

Example snippet from MIT license:

```
swift
CopyEdit
MIT License
```

```
Copyright (c) 2024 Laura Mueller
```

```
Permission is hereby granted, free of charge, to any person
obtaining a copy...
```

3. Creating a README File

- The README is the **first contact point** for users.
- Should contain:
 - Brief description of what the module does.
 - Main features (e.g., manages VPC and public/private subnets).
 - Simple example usage snippet.
 - Optionally, inputs and outputs descriptions or links.
 - Notes on how to configure key options.

Example README Structure:

```
markdown
CopyEdit
# Networking Module
```

```
This module manages the creation of VPCs and subnets, supporting
both public and private subnets.
```

```

## Example Usage

```terraform
module "vpc" {
 source = "./networking"

 vpc_name = "your-vpc"
 cidr_block = "10.0.0.0/16"

 subnet_config = {
 subnet1 = {
 cidr_block = "10.0.1.0/24"
 az = "eu-west-1a"
 public = false
 }
 subnet2 = {
 cidr_block = "10.0.2.0/24"
 az = "eu-west-1b"
 public = true
 }
 }
}
```

```

4. Examples Folder Structure

- Create an `examples/` directory inside your module root.
- Organize examples by use cases or complexity, e.g.,:

```

css
CopyEdit
networking/
|__ examples/
|   |__ complete/
|   |   |__ main.tf
|   |__ minimal/
|   |   |__ main.tf

```

- Each example is a self-contained Terraform configuration demonstrating how to use the module.
 - This helps users quickly understand and test the module.
-

Inside `examples/complete/main.tf`

- Copy your typical or full usage of the module here.
- Add comments explaining key parameters:

```
hcl
CopyEdit
# Mark public subnets by setting `public = true`
module "vpc" {
  source = "../../"

  vpc_name    = "example-vpc"
  cidr_block = "10.0.0.0/16"

  subnet_config = {
    subnet1 = {
      cidr_block = "10.0.1.0/24"
      az         = "eu-west-1a"
      public     = false
    }
    subnet2 = {
      cidr_block = "10.0.2.0/24"
      az         = "eu-west-1b"
      public     = true  # This is a public subnet
    }
  }
}
```

5. Why These Are Important

| Aspect | Purpose |
|--------|---------|
|--------|---------|

| | |
|----------|--|
| License | Clarifies legal permissions and protects author & users |
| README | Provides documentation and quick-start guidance |
| Examples | Practical demos improve usability and lower the learning curve for new users |

6. Final Tips

- Choose license carefully based on:
 - How you want your module to be used.
 - Whether it's open source or proprietary.
- README should be detailed if module is public or widely used.
- Provide multiple examples if your module has complex or varied use cases.
- Keep example code simple, commented, and runnable as-is.

Lecture 15: Testing the Module by Creating EC2 Instances & Adding Subnet Tags

1. Purpose

- Validate the networking module by **deploying EC2 instances** using subnet outputs.
 - Practice **referencing outputs from your module** to wire up dependent resources.
 - Enhance subnet resource tagging to differentiate public/private subnets clearly.
-

2. Setup: Create `compute.tf` in Root Module

- Create a new Terraform file, e.g., `compute.tf`.

Use existing data sources for AMI, such as:

```
hcl
CopyEdit
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical Ubuntu
  filter {
    name   = "name"
    values =
    ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

•

Create an AWS EC2 instance referencing the subnet from your networking module outputs:

```
hcl
CopyEdit
resource "aws_instance" "example" {
```

```
ami          = data.aws_ami.ubuntu.id
instance_type = "t3.micro"
subnet_id     = module.vpc.private_subnets["subnet_1"].subnet_id

tags = {
  Name = local.project_name
}
}
```

•

Define a local variable for project name to keep tags consistent:

```
hcl
CopyEdit
locals {
  project_name = "13_local_modules"
}
```

•

3. Referencing Module Outputs

- Use outputs defined in your VPC module to get subnet IDs.
- Note the exact output attribute names (e.g., `subnet_id`).

Example:

```
hcl
CopyEdit
subnet_id = module.vpc.private_subnets["subnet_1"].subnet_id
```

•

4. Terraform Workflow

- Run `terraform plan` to validate the config.

- Run `terraform apply` to deploy the EC2 instance in the specified subnet.
 - Check the AWS Console → EC2 → Instances to confirm running instance.
-

5. Improvement: Tagging Subnets with Access Type

- Enhance subnet resource to **tag each subnet** with `access = "public"` or `access = "private"` based on the `public` boolean property.

Example subnet tagging block:

```
hcl
CopyEdit
tags = merge(
  var.common_tags,
  {
    Name    = each.key
    Access = each.value.public ? "public" : "private"
  }
)
```

- Benefits:
 - Easier filtering and identification of subnet types in AWS Console.
 - Simplifies infrastructure management and scripting.

6. Apply Tagging Changes

- Run `terraform apply` again to update subnet tags.
 - Verify tags in AWS Console → VPC → Subnets.
-

7. Clean Up

- To avoid costs (especially with EC2 instances):
 - Run `terraform destroy` to remove all resources.
 - Confirm all resources (instances, subnets, VPC) are deleted.
-

Summary

| Step | Details |
|---------------------|--|
| Create EC2 resource | Use module outputs for subnet IDs, specify AMI and instance type |
| Reference outputs | Access subnet IDs using keys from module outputs (maps for stability) |
| Tag subnets | Add <code>Access = public/private</code> tag based on subnet property |
| Validate deployment | <code>terraform plan</code> → <code>terraform apply</code> → verify in AWS Console |
| Clean up resources | Use <code>terraform destroy</code> to avoid charges |

Lecture 16: Publishing Terraform Modules

1. Why Publish a Terraform Module?

- Sharing your module with others publicly or internally.
 - Enables **versioning** to manage breaking changes smoothly.
 - Provides **documentation**, **examples**, and **history** automatically.
 - Easier reuse across projects, teams, or organizations.
 - Public registry is great for open-source modules; private registry suits company-internal usage.
-

2. Requirements for Publishing on the Public Terraform Registry

- Module **must be hosted on GitHub** as a **public repository**.

Repository name must follow the convention:

php-template
CopyEdit
terraform-<PROVIDER>-<MODULE_NAME>

- Example: **terraform-aws-networking**
 - Add a **description** to the GitHub repo; this populates the module's short description on the Terraform registry.
-

3. Module Structure Standards

- Use **standard module structure**:
 - `variables.tf` for input variables.
 - `outputs.tf` for output variables.
 - `main.tf` for resources.
 - `README.md` describing usage.
 - Optional: examples folder, license file.
 - This structure helps the Terraform registry to auto-generate **documentation** and makes the module easier to use.
-

4. Versioning with Semantic Release

- Use **tags** in GitHub with semantic versioning format: `vX.Y.Z`
 - `X` = major version (breaking changes)
 - `Y` = minor version (new features, backward compatible)
 - `Z` = patch version (bug fixes, no new features)
 - Version tags let consumers select and pin module versions safely.
-

5. Why Not Just Develop Locally?

- Local modules lack version control, so breaking changes are harder to manage.
 - Published modules enable **independent updates** without breaking dependent projects immediately.
 - Allows browsing previous versions and documentation online.
-

6. Private Registries

- If your module is for internal use only:
 - Publish on a **private Terraform registry**.
 - GitHub can host private repos, but you need private registry support.
 - Get benefits like versioning, documentation, and controlled access.
-

7. Summary

| Topic | Key Points |
|---------------------|--|
| Publishing location | Public GitHub repo (public registry) or private repo/registry (internal use) |
| Repo naming | <code>terraform-<provider>-<module></code> |
| Documentation | Add description on GitHub + README + variables/outputs files for docs |
| Versioning | Use semantic versioning tags (<code>vX.Y.Z</code>) |
| Benefits | Versioning, easier updates, docs, sharing, independent maintenance |

This lecture sets the stage for the **next steps** — actually publishing your module to the Terraform registry.

Lecture 17: Publishing Terraform Module — Step-by-Step

1. Create GitHub Repository

Naming convention:

CopyEdit
`terraform-aws-networking`

-
- Add **description** (used by Terraform Registry for module summary).
- Set repository to **public** (mandatory for public Terraform registry).
- Add:
 - `README.md` file (module documentation)
 - `.gitignore` (Terraform template)
 - License file (MIT or any permissive license).

2. Clone Repository Locally

- Clone into a **new folder** (avoid nesting in existing git repos to prevent conflicts).
 - Open folder in IDE (e.g., VS Code).
 - Copy existing module files (`variables.tf`, `outputs.tf`, `main.tf`, `examples/`, etc.) into this new repo.
 - Update the README with relevant info and module description.
-

3. Commit and Push

- Stage and commit all files.
 - Push to remote GitHub repo.
 - Initially, no **tags** or releases exist.
-

4. Tagging for Versioning

Use semantic versioning tags, e.g.,

```
css
CopyEdit
git tag v0.1.0
git push origin --tags
```

- - Use **0.x.y** for early development versions before stable 1.0.0 release.
 - Tagging is crucial for Terraform Registry version control.
-

5. Publish Module on Terraform Registry

- Sign in to Terraform Registry with GitHub.
- Grant access to your GitHub repositories.
- Click **Publish Module**.
- Registry lists eligible repositories based on naming convention.
- Select your module repo and accept terms.
- Publish the module.
- Terraform Registry reads your module's structure and auto-generates documentation:
 - Inputs (variables)
 - Outputs

- Usage instructions (README)
 - Examples folder is recognized (submodules shown in UI).
-

6. Improve Variable Descriptions

- Variables should have **descriptions** for better documentation.

Use multi-line descriptions with Heredoc syntax for readability if needed.

Example for `subnet_config` variable:

```
hcl
CopyEdit
description = <<EOT
Map of subnet configs. Each contains:
- cidr_block: The subnet CIDR block
- public: Boolean to mark if public subnet (default false)
- az: Availability Zone to deploy the subnet
EOT
```

- Providing descriptions improves the Registry UI and usability.
-

7. Push Updates and Tag Patch Versions

- After improvements, commit and push changes.
 - Tag a new patch version (`v0.1.1`) for non-breaking fixes or documentation updates.
 - Push tags so the registry updates module versions.
-

8. Using Your Published Module

- Terraform Registry provides the exact usage snippet.
- Create a new Terraform project folder.

Use the module by referencing the registry source with a version:

```
hcl
CopyEdit
module "networking" {
  source  = "your-github-username/networking/aws"
  version = "0.1.1"

  vpc_config = {...}
  subnet_config = {...}
}
```

- Run `terraform init` and `terraform plan` to verify usage.

9. Best Practices & Recommendations

- Use **CI/CD pipelines** to automate module publishing & tagging instead of manual steps.
- Keep modules well-documented with clear inputs, outputs, and examples.
- Use semantic versioning strictly to signal breaking and non-breaking changes.
- Balance detailed variable descriptions with usability — avoid overly verbose docs if possible.

Summary Table

| Step | Key Points |
|--------------------|---|
| GitHub Repo Setup | Naming convention, public repo, add README, license, .gitignore |
| Clone & Copy Files | Avoid nested git repos, organize files properly |

| | |
|------------------------------|---|
| Commit & Push | Initial commit, push code |
| Tagging | Use semantic versioning, start with 0.1.0 for early dev |
| Publish on Registry | Authenticate, select repo, publish, auto-documentation |
| Improve Documentation | Add variable descriptions, use multi-line format for clarity |
| Version Updates | Commit fixes, tag patch versions, push tags |
| Use Module | Copy usage snippet from registry, <code>terraform init</code> & <code>plan</code> |
| Automation
Recommendation | Use CI/CD pipelines for publishing |

Object Validation

Terraform Object Validation Overview

Why Object Validation?

- Ensures **more reliable infrastructure** by validating inputs and configurations.
 - Supports **preconditions, postconditions, and check assertions** to verify correctness at different stages.
 - Helps catch errors **before** or **after** resource creation, or simply warns users without blocking deployment.
-

1. Preconditions

- Defined inside the **lifecycle** block of a resource or data source.
- **Cannot reference the resource itself** (no access to resource attributes yet).
- Operate only on **external data or inputs** (e.g., variables or data blocks the resource depends on).
- Example use case: Check if a variable or data block is valid before resource creation.

Syntax:

```
hcl
CopyEdit
lifecycle {
  precondition {
    condition      = <boolean expression>
    error_message = "Error if condition false"
  }
}
```

-
- If **condition** evaluates to **false**, Terraform **stops** with the error message.

- Useful for **early validation** of inputs.
-

2. Postconditions

- Also defined inside **lifecycle** block of resources or data blocks.
- **Can reference the resource itself** via the special keyword **self**.
- Validate the state **after resource creation** or during apply.

Syntax:

```
hcl
CopyEdit
lifecycle {
  postcondition {
    condition      = <boolean expression involving self>
    error_message = "Error if condition false"
  }
}
```

- Example: Check that an attribute of the created resource matches expectations.
 - Also **stops Terraform apply** if condition fails.
 - Useful for **ensuring resource correctness**.
-

3. Check Assertions (**check** blocks)

- Introduced in recent Terraform versions.
- Defined **outside** of resources and data sources.
- Can reference **any information in the Terraform configuration/project**.

Syntax:

```

hcl
CopyEdit
check {
  condition      = <boolean expression>
  error_message = "Warning message"
}

```

- - **Do not stop Terraform apply**; only emit **warnings**.
 - Use case example:
 - Warn if resources are missing optional but recommended tags.
 - Good for **non-critical validations** where you want awareness but no enforcement.
-

Summary Table

| Validation Type | Location in Code | Can Reference Resource? | Effect if Condition Fails | Typical Use Case |
|------------------|------------------------------|---------------------------|---------------------------|--|
| Preconditions | <code>lifecycle</code> block | No | Stops apply, shows error | Validate inputs before resource creation |
| Postconditions | <code>lifecycle</code> block | Yes (<code>self</code>) | Stops apply, shows error | Validate resource after creation |
| Check assertions | Outside resource blocks | Yes | Warning only, no stop | Warn about non-critical issues |

Key Points

- Preconditions and postconditions are **strict validations** that prevent deployment if failed.
- Check assertions are **soft validations** — just warnings.
- Use preconditions for **input validation**.

- Use postconditions for **resource state validation**.
 - Use check assertions for **best practices and style warnings**.
 - Balancing these improves **infrastructure reliability** and **user feedback**.
-

What's Next?

- Hands-on practical exercises to implement each validation type.
- Deep dive into syntax and usage examples.
- Learn how to write meaningful conditions and error messages.

Lecture 2

Terraform Object Validation: Preconditions Deep Dive

Overview

- We explore **preconditions**, a powerful way to validate resource input **dynamically**.
 - Preconditions extend beyond simple variable validation by allowing references to **external data sources** or variables.
 - Preconditions **do not** have access to the resource's own attributes (`self` is **not allowed**).
 - They run **after** `for_each` and `count` meta-arguments are evaluated, enabling dynamic iteration validations.
-

Setting Up

- Create a new Terraform folder for the exercise, e.g., [15-Object-Validation](#).
 - Define providers and required versions (e.g., AWS provider version 5.0).
 - Precondition blocks live inside the **lifecycle block** of a resource or data block.
-

Why Preconditions Over Variable Validation?

| Variable Validation | Preconditions |
|--|--|
| Limited to variable's own value only (no external references). | Can reference external data sources , variables, or dynamic data. |
| Hardcoded or static checks. | Can perform dynamic, iteration-based validations (<code>for_each</code> support). |

Stops execution if validation fails.

Stops execution if condition fails with custom error message.

Key Limitation of Preconditions

- Cannot reference resource's own attributes (`self`) — resource not fully created yet.
 - Therefore, preconditions are ideal for checking `inputs` or `external dependencies`, not validating the resource's created state.
-

Example: Validate Allowed Instance Types

Variables Definition (`variables.tf`)

```
hcl
CopyEdit
variable "instance_type" {
  type    = string
  default = "t2.micro"
}
```

Locals for Allowed Types

```
hcl
CopyEdit
locals {
  allowed_instance_types = ["t2.micro", "t3.micro"]
}
```

Resource with Precondition (`compute.tf`)

```
hcl
CopyEdit
resource "aws_instance" "example" {
  ami          = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  lifecycle {
```

```

precondition {
    condition      = contains(local.allowed_instance_types,
var.instance_type)
    error_message = "Invalid instance type: ${var.instance_type}.
Allowed: ${local.allowed_instance_types}."
}
}
}

```

- If `var.instance_type` is NOT in `allowed_instance_types`, Terraform **stops apply** with a clear error.
 - If valid, apply proceeds normally.
-

What Happens If You Try to Reference `self` in Preconditions?

Example invalid precondition:

```

hcl
CopyEdit
lifecycle {
  precondition {
    condition = self.instance_type == "t2.micro"
    error_message = "Invalid instance type"
  }
}

```

- This will **fail** during plan/apply with an error: `self` cannot be referenced in preconditions.
 - Because the resource is not yet created when preconditions run.
-

Summary

Concept

Explanation

| | |
|-----------------|---|
| Preconditions | Validate input data or external references before creation. |
| Purpose | |
| Location | Inside resource lifecycle block. |
| Resource Access | No access to resource attributes (self not allowed). |
| When to Use | Validate variables, external data, or parameters used in resource creation. |
| Outcome | If validation fails, Terraform stops with an error message. |

Best Practice Tips

- Use preconditions to **fail fast** on invalid inputs.
 - Combine with variable validation for extra safety.
 - For validating the resource's **own attributes**, use **postconditions** (covered next lecture).
 - Craft **clear error messages** to guide users on fixing inputs.
-

Next Steps

- Upcoming lecture: **Postconditions**, how to validate resource attributes **after creation**.
- Exercises to practice preconditions and postconditions together.

Terraform Lecture 3: Postconditions — GOD LEVEL NOTES

1. What are Postconditions?

- Postconditions are **validation blocks inside the lifecycle block of a resource or data block**.
 - They **validate the resource after it has been created or modified**.
 - Postconditions can access the resource's own attributes using `self` (e.g., `self.instance_type`).
 - They can run:
 - **During plan phase** if all necessary data is known upfront.
 - **During apply phase** if some resource attributes (e.g., `subnet_id`, `availability_zone`) are only assigned after creation.
 - If a postcondition fails, Terraform **halts the apply and stops creating dependent resources**.
-

2. Why Use Postconditions?

- To **enforce resource constraints that depend on actual resource state**.
 - To validate dynamic attributes assigned by AWS or other providers after resource creation.
 - To **prevent deployment of resources with invalid configurations** even if input variables appear valid.
 - To **avoid cascading failures** by stopping downstream resources when the current resource is invalid.
-

3. Difference Between Preconditions and Postconditions

| Aspect | Preconditions | Postconditions |
|--------------------------------------|---|---|
| Where defined | <code>lifecycle</code> block of resource or data source | <code>lifecycle</code> block of resource or data source |
| Access to resource attributes | NO (cannot use <code>self</code>) | YES (can reference <code>self</code>) |
| When evaluated | During plan phase only | During plan or apply phase depending on data availability |
| Effect on apply | Stops apply if condition fails | Stops apply if condition fails |
| Use case | Validate inputs or external data | Validate actual resource state |

4. Setup Example — Variables and Locals

```
hcl
CopyEdit
variable "instance_type" {
    type    = string
    default = "t2.micro"
}

locals {
    allowed_instance_types = ["t2.micro", "t3.micro"]
}
```

- `instance_type` is a user input variable.
 - `allowed_instance_types` is a local list for validation.
-

5. Data Sources (AMI and VPC)

```
hcl
CopyEdit
data "aws_ami" "ubuntu" {
```

```

most_recent = true
owners      = ["099720109477"] # Canonical official AMI owner ID
filter {
  name   = "name"
  values =
  ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
}
}

data "aws_vpc" "default" {
  default = true
}

```

- Fetches latest Ubuntu AMI.
 - Fetches default VPC for subnet creation.
-

6. Subnet Resource with Postcondition

hcl
[CopyEdit](#)

```

resource "aws_subnet" "example" {
  vpc_id      = data.aws_vpc.default.id
  cidr_block = "172.31.128.0/24" # Must not conflict with existing
                                subnets

  lifecycle {
    postcondition {
      # Simple check to ensure CIDR block is as expected
      condition      = cidr_block == "172.31.128.0/24"
      error_message = "Subnet CIDR block is invalid or conflicts
with existing subnet."
    }
  }
}

```

- Postcondition validates that subnet CIDR block is exactly the one intended.

- If invalid (e.g., conflicts with existing subnet), this fails *after* apply if subnet creation is deferred.
-

7. AWS Instance Resource with Multiple Postconditions

```

hcl
CopyEdit
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.example.id

  lifecycle {
    # Validate instance type is allowed (can run at plan phase)
    postcondition {
      condition      = contains(local.allowed_instance_types,
self.instance_type)
      error_message = "Invalid instance type: ${self.instance_type}.

Allowed: ${local.allowed_instance_types}."
    }

    # Validate availability zone is among available zones (known
only after apply)
    postcondition {
      condition      =
contains(data.aws_availability_zones.available.names,
self.availability_zone)
      error_message = "Invalid availability zone:
${self.availability_zone}."
    }

    # Create the new resource before destroying the old one to
prevent downtime
    create_before_destroy = true
  }
}

```

Explanation:

- `self.instance_type` can be validated immediately because this is a user-set input and known at plan.
 - `self.availability_zone` is assigned by AWS on creation, so this check can only run *after* apply.
 - `create_before_destroy = true` helps prevent downtime during resource replacement.
-

8. Why is `create_before_destroy` Important Here?

- Without it, Terraform destroys the old resource before creating a new one.
 - If postcondition fails on new resource creation, you end up with **no running instance (downtime)**.
 - With `create_before_destroy = true`, Terraform creates the new resource first, **validates postconditions**, then destroys the old one **only if postconditions pass**.
-

9. Plan and Apply Phase Behavior

- At **plan phase**, Terraform knows static inputs and data sources.
 - Postconditions that depend only on this info are **evaluated during plan**.
 - Postconditions that depend on **runtime-assigned attributes** (like subnet ID, AZ) are **evaluated after apply**.
 - Failure during apply stops further resource creation and rollbacks occur as needed.
-

10. Example: Invalid Availability Zone Check

hcl

CopyEdit

```
lifecycle {  
  postcondition {  
    condition      = self.availability_zone == "eu-central-1a"
```

```

        error_message = "Invalid availability zone:
${self.availability_zone}."

    }
}

```

- If the configured region is `eu-west-1`, the above will always fail.
 - This postcondition won't fail at plan phase because `availability_zone` is unknown.
 - After apply, Terraform will fail and stop further downstream resources.
-

11. Full Working Example

```

hcl
CopyEdit
variable "instance_type" {
  type    = string
  default = "t2.micro"
}

locals {
  allowed_instance_types = ["t2.micro", "t3.micro"]
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]
  filter {
    name    = "name"
    values =
    ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

data "aws_vpc" "default" {
  default = true
}

resource "aws_subnet" "example" {

```

```

vpc_id      = data.aws_vpc.default.id
cidr_block = "172.31.128.0/24"

lifecycle {
  postcondition {
    condition      = cidr_block == "172.31.128.0/24"
    error_message = "Subnet CIDR block invalid or conflicts."
  }
}
}

resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.example.id

  lifecycle {
    postcondition {
      condition      = contains(local.allowed_instance_types,
self.instance_type)
      error_message = "Invalid instance type:
${self.instance_type}."
    }
    postcondition {
      condition      =
contains(data.aws_availability_zones.available.names,
self.availability_zone)
      error_message = "Invalid availability zone:
${self.availability_zone}."
    }
    create_before_destroy = true
  }
}

```

12. Commands to Try

bash
CopyEdit
terraform init
terraform fmt -recursive

```
terraform plan  
terraform apply  
terraform destroy
```

13. Summary — Important Takeaways

- **Postconditions validate the final state of resources and can access `self`.**
- They run either during plan or apply depending on attribute availability.
- Use postconditions to **ensure infrastructure correctness and compliance dynamically**.
- Combine with `create_before_destroy` for safe resource replacements.
- Always write **clear error messages** for easier debugging.
- Remember: **Preconditions are for input validation; postconditions are for resource validation.**

Lecture 4: When Are Preconditions and Postconditions Executed? — GOD LEVEL NOTES

1. The Big Picture: Validation Happens in Two Phases

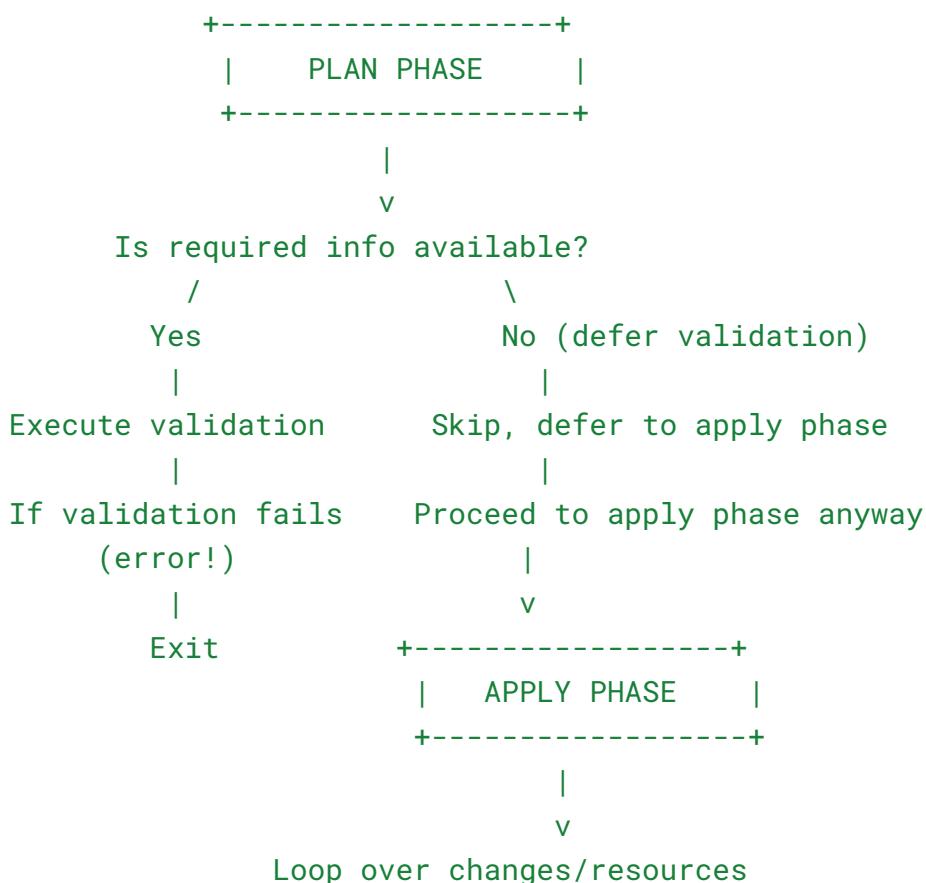
Terraform runs validations (preconditions & postconditions) **in two main phases**:

- **Plan Phase** (`terraform plan` or the planning stage of `terraform apply`)
 - **Apply Phase** (`terraform apply` execution)
-

2. Validation Execution Flow Diagram (Conceptual)

vbnnet

CopyEdit





3. What Happens in the Plan Phase?

- Terraform **examines all resources and data** needed to create/update/destroy infrastructure.
 - For validations that depend only on data already known, Terraform **executes those validations immediately**.
 - If any validation fails here, Terraform exits with an error — no apply happens.
 - For validations that require data not yet available (e.g., attributes assigned after resource creation), Terraform defers those checks to the apply phase.
 - **Important:** A successful plan phase does not guarantee all validations passed, only those that could be checked.
-

4. What Happens in the Apply Phase?

- Terraform starts applying changes resource by resource.
- For each resource or change:
 - Terraform executes any deferred validations (postconditions) that require data generated by the apply.
 - If validation passes, Terraform moves to the next resource/change.
 - If validation fails, Terraform stops immediately, returning an error.
- This can lead to **partial application**: all changes before the failure are kept; changes after the failure are not applied.

- You must **fix the validation error and re-apply** to finish applying all changes.
-

5. Why is This Important?

- You might **see a successful `terraform plan` but `terraform apply` fails** due to deferred validations.
 - This can be confusing at first because:
 - Plan phase validation only works with known information.
 - Apply phase validation happens when actual resource attributes are assigned.
 - Infrastructure **may be partially deployed** before a failure occurs during apply.
 - This means **Terraform does not automatically roll back changes made during `apply`** before the failure.
 - You need to **address errors and run apply again**.
-

6. Dynamic Availability of Information

- What information Terraform has at plan phase **depends on your state and configuration**.
 - As you **add or modify resources**, more information becomes available to Terraform upfront.
 - Over time, some deferred validations can become plan-time validations.
 - This means the behavior of when validations run **may change as your infrastructure grows**.
-

7. Practical Implications & Tips

- Always **write clear, meaningful error messages** in your pre/postconditions.
 - Expect some validations to happen during apply, especially those dependent on resource attributes like:
 - `availability_zone`
 - `subnet_id`
 - dynamically assigned IDs or metadata
 - Use `terraform plan` to catch early validation errors — but be ready for apply phase errors.
 - If apply errors occur, **fix the root cause** before retrying.
 - Consider enabling `create_before_destroy = true` on critical resources to reduce downtime if replacements happen.
-

8. Summary Table: Validation Execution

| Validation Type | When Runs | Can Reference <code>self</code> ? | Failure Effect |
|---------------------------------|--------------------------------|-----------------------------------|--|
| Variable validation | Plan phase | No | Stops plan/apply immediately |
| Preconditions | Plan phase (if info available) | No | Stops plan/apply immediately |
| Postconditions | Plan phase or apply phase | Yes | Stops apply immediately or after apply |
| Check assertions (warning only) | Plan or apply | No | Shows warning but does NOT stop apply |

9. Final Takeaway

- **Plan phase validations are "early warnings" for issues with known data.**

- **Apply phase validations are the final gatekeepers for resource correctness based on real state.**
- Always test your Terraform workflows with both `plan` and `apply` to catch all errors.

Lecture 5: Terraform Check Blocks — God Level Notes

1. What Are Check Blocks?

- **Check blocks** in Terraform are used for **non-critical validations**.
 - Unlike **preconditions** and **postconditions** that fail and stop Terraform execution if violated, **checks only emit warnings**.
 - They **do NOT stop the apply or plan process**.
 - Perfect for **advisory warnings** about potential misconfigurations or suboptimal infrastructure setups.
-

2. Why Use Check Blocks?

- To **inform users** or your team of:
 - Missing but recommended tags or metadata.
 - Suboptimal architectural patterns (e.g., not distributing subnets across AZs).
 - Potential risks without halting deployment.
 - Helps **improve infrastructure hygiene** gradually without blocking deployments.
-

3. Check Block Syntax Overview

```
hcl
CopyEdit
check "check_name" {
  assert      = <condition_expression>
  error_message = <<EOT
  Multi-line warning message
  EOT
```

```
}
```

- **assert**: Boolean expression, if `false`, emits a **warning**.
 - **error_message**: Friendly message shown when assertion fails.
-

4. Example 1: Checking for a Cost Center Tag on an AWS Instance

You want to warn users if they **forgot to tag an EC2 instance** with a `cost_center` tag.

Terraform snippet:

```
hcl
CopyEdit
resource "aws_instance" "example" {
  # ... your instance config ...

  check "cost_center_check" {
    assert = can(self.tags.cost_center) && self.tags.cost_center != ""
    error_message = "Warning: Your AWS instance does not have a 'cost_center' tag."
  }
}
```

Explanation:

- `can(self.tags.cost_center)` returns `false` if the tag doesn't exist, avoiding runtime errors.
 - `self.tags.cost_center != ""` ensures the tag is not empty.
 - If either fails, Terraform will show a **warning** during `plan` and `apply`.
 - **Apply still proceeds normally**, user is just warned.
-

5. Running the Example

- **Without the tag:** Terraform prints the warning but applies the resource.
- **Add the tag:**

```
hcl
CopyEdit
tags = {
  cost_center = "1234"
}
```

- Re-running plan/apply: Warning disappears.
-

6. Example 2: Check for High Availability Subnet Deployment

Warn users if **all subnets are deployed in the same Availability Zone (AZ)**, which is a bad HA practice.

Setup:

- Create multiple subnets using `count`.
- Assign the **same AZ** to all subnets.

Terraform snippet:

```
hcl
CopyEdit
resource "aws_subnet" "this" {
  count = 2

  vpc_id          = data.aws_vpc.default.id
  cidr_block      = "172.31.${count.index + 1}28".0/24"
  availability_zone = "eu-west-1a" # All in the same AZ for this
example

  check "high_availability_check" {
```

```

    assert = length(distinct([for subnet in aws_subnet.this :
subnet.availability_zone])) > 1
    error_message = <<EOT
Warning: You are deploying all subnets within the same Availability
Zone.
Please consider distributing them across multiple AZs for higher
availability.
EOT
}
}

```

Explanation:

- `for subnet in aws_subnet.this : subnet.availability_zone` collects all AZs from created subnets.
 - `distinct(...)` extracts unique AZs.
 - `length(...) > 1` checks if subnets span more than one AZ.
 - If `false` → Terraform warns, but continues.
-

7. Fixing the High Availability Warning: Use Round-Robin AZ Assignment

Instead of hardcoding the AZ, use modulo (%) to assign AZs dynamically in a round-robin fashion:

```

hcl
CopyEdit
availability_zone =
data.aws_availability_zones.available.names[count.index %
length(data.aws_availability_zones.available.names)]

```

- This rotates AZs across all available zones.
 - Removes the warning.
-

8. Summary Table: Precondition vs Postcondition vs Check

| Feature | Stops Execution on Fail? | When Runs | Use Case |
|---------------|--------------------------|---|--|
| Precondition | Yes | Plan Phase (if info available) | Critical validation before apply |
| Postcondition | Yes | Apply Phase (or plan if info available) | Critical validation on resource attributes |
| Check | No (warn only) | Plan and Apply | Non-critical advisory checks |

9. Practical Advice

- Use **check blocks** for best practices & soft warnings.
- Use **pre/postconditions** for must-have, critical validation.
- Check blocks improve **infrastructure quality without risking deployment failures**.
- Customize error messages to be **clear, helpful, and actionable**.

10. Bonus: Terraform Format & Plan

Always run:

```
bash
CopyEdit
terraform fmt -recursive
terraform plan
```

to:

- Ensure readable and consistent code.
- See warnings and validation feedback.

11. Cleaning Up: Destroy Resources

After your experiments, destroy all deployed resources with:

```
bash
CopyEdit
terraform destroy
```

to avoid unnecessary cloud charges.

End of Lecture 5 God Level Notes

State Manipulation

Terraform — Lecture 1: State Manipulation (God-Level Notes)

0. Context

Terraform's *state file* is the single source of truth for what Terraform thinks exists in your infrastructure.

State manipulation is **directly altering Terraform's perception** of your resources **without necessarily touching the real infrastructure**.

This is **not casual use** — these commands are for **fixing, importing, refactoring, and recovering** infrastructure.

1. What is State Manipulation?

Definition

State manipulation is the process of **managing, altering, or syncing** Terraform's state **outside the normal `plan` → `apply` cycle**, in order to:

- Fix broken resources without config changes.
 - Import existing resources under Terraform.
 - Refactor resource names/modules without recreation.
 - Remove resources from Terraform without deleting them.
 - Resolve stuck states or mismatched states.
-

2. Why Do We Need State Manipulation?

Terraform's behavior is **state-driven**:

1. Reads **HCL config**.

2. Reads **state file** (local/remote backend).
3. Compares → determines changes.

If **state** and **reality** get out of sync, Terraform will make incorrect changes:

- Might destroy working infra.
- Might fail to detect issues.
- Might refuse to manage existing infra.

State manipulation solves:

- Infra corruption without config changes.
 - Adopting pre-existing resources.
 - Avoiding destructive recreations during refactoring.
 - Handing over resources between teams.
 - Cleaning Terraform's state without destroying cloud infra.
-

3. Core Use Cases & Commands

3.1 Tainting

Purpose: Force Terraform to **recreate a resource** without changing config.

When to use:

- Resource is unhealthy/corrupted.
- Config is fine, but resource needs rebuilding.

Command:

```
bash
CopyEdit
terraform taint aws_instance.my_instance
```

Effect:

- Marks `aws_instance.my_instance` as *tainted* in state.
- On next `terraform apply`, it will **destroy** → **recreate** the resource.

Undo:

```
bash
CopyEdit
terraform untaint aws_instance.my_instance
```

📌 **Key Point:** No `.tf` changes needed — taint exists only in the state.

3.2 Importing

Purpose: Adopt an **existing resource** into Terraform management.

Command:

```
bash
CopyEdit
terraform import aws_instance.my_instance i-0abc1234def5678gh
```

Explanation:

- `aws_instance.my_instance` → The name in your `.tf` file.
- `i-0abc1234def5678gh` → The actual AWS resource ID.

📌 **Steps for a safe import:**

1. Create `.tf` definition for the resource.
2. Run `terraform import`.
3. Run `terraform plan` to check for diffs.
4. Adjust `.tf` to match actual resource config.

⚠️ **Warning:** Import does not modify the real infra — only updates Terraform state.

3.3 Refactoring (Moving/Renaming Resources)

Problem:

Renaming resource in `.tf` triggers destroy+create.

Solution:

bash

CopyEdit

```
terraform state mv aws_subnet.old_name aws_subnet.new_name
```

Explanation:

- Changes **address in state file**.
- Infra remains untouched.
- Terraform now maps the existing resource to the new name.

 **Use Case:**

- Moving resources into modules.
- Standardizing naming without downtime.

3.4 Untracking (Remove from State Without Deleting)

Purpose: Stop managing a resource **without destroying it**.

Command:

bash

CopyEdit

```
terraform state rm aws_instance.my_instance
```

Effect:

- Resource is removed from state.
- Stays in cloud.

- Terraform no longer manages it.

📌 **Typical Use Case:**

Handing over infra ownership to another team.

3.5 Generating Configuration from Existing Resources

Purpose: Let Terraform create `.tf` from an existing resource.

Command:

```
bash
CopyEdit
terraform plan -generate-config-out=generated.tf
```

Notes:

- Works with `terraform import`.
 - Produces a **best effort** config — will require manual refinement.
-

3.6 Special State Operations

Force Unlock

If a state lock gets stuck (e.g., crash during apply):

```
bash
CopyEdit
terraform force-unlock LOCK_ID
```

Pull Remote State

```
bash
CopyEdit
terraform state pull > state.json
```

Use when:

- You want to inspect the raw state.

- Debugging differences.

Push State Back

bash

CopyEdit

```
terraform state push state.json
```

 Danger — overwrites backend state. Only for emergencies.

4. Internal Mechanics — How These Commands Work

- Terraform **state** file stores:
 - Resource addresses.
 - Metadata (IDs, dependencies).
 - Outputs.
- State manipulation commands **only modify this mapping** — not `.tf` files or cloud infra.
- On next `plan`, Terraform:
 - Uses modified state.
 - Decides what to create/destroy/update.

 **Analogy:** The state file is Terraform's **map of your infrastructure**. Changing the state is like **relabeling the map**, not moving the real terrain.

5. Best Practices for State Manipulation

- **Always backup state** before manipulation:

bash

CopyEdit

```
terraform state pull > backup.json
```

- Use **remote backends** (S3, Terraform Cloud) for safety.
 - Minimize direct manipulation — prefer config changes unless downtime risk is high.
 - Document every manipulation — future maintainers need context.
 - Never hand-edit `terraform.tfstate` unless commands can't solve it.
-

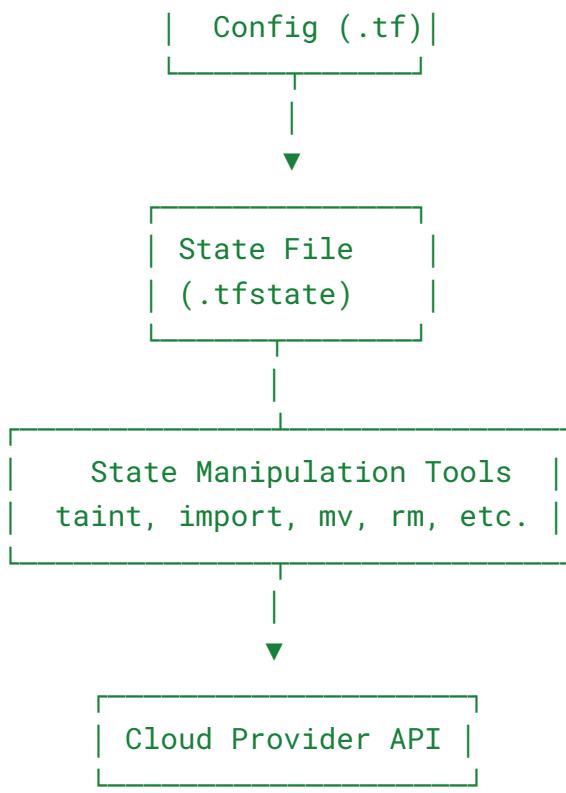
6. Summary Table of State Commands

| Command | Purpose | Destroys Infra? | Changes Config? | Changes State? |
|-------------------------------------|-------------------------------|-----------------|-----------------|------------------|
| <code>terraform taint</code> | Force recreate on next apply | ✓ (next apply) | ✗ | ✓ |
| <code>terraform untaint</code> | Remove taint mark | ✗ | ✗ | ✓ |
| <code>terraform import</code> | Adopt existing resource | ✗ | ✗ | ✓ |
| <code>terraform state mv</code> | Rename/move resource in state | ✗ | ✗ | ✓ |
| <code>terraform state rm</code> | Stop tracking resource | ✗ | ✗ | ✓ |
| <code>terraform force-unlock</code> | Release stuck lock | ✗ | ✗ | ✓ |
| <code>terraform state pull</code> | Download state | ✗ | ✗ | ✗
(read-only) |
| <code>terraform state push</code> | Upload state | ✗ | ✗ | ✓ |

7. Visual Workflow — State Manipulation

arduino
CopyEdit





8. Mental Model to Remember

Think of Terraform's state file as **the ledger** in a bank.

If the ledger says you have \$100, Terraform believes you do — even if reality is different.

State manipulation is editing the ledger without touching the actual vault.

Lecture 2 – Terraform State Manipulation, Refactoring & Moved Blocks

1. Overview

This lecture explores **refactoring Terraform configurations** without destroying infrastructure, using:

1. **CLI-based state manipulation** (`terraform state mv`)
2. **Configuration-based moved blocks**

Also covers:

- Why renaming resources can cause unwanted destroys/recreates
- Handling changes with **count** and **for_each**
- Preserving resource history when refactoring
- Moving resources **into/out of modules**

2. Key Definitions

| Term | Meaning |
|--|---|
| Terraform State File | A file (<code>terraform.tfstate</code>) that tracks the mapping between Terraform config resources and real cloud resources. |
| Resource Address | The unique identifier in Terraform's state, composed of the resource type + local name (label). E.g., <code>aws_instance.my_instance</code> . |
| Moved Command
(<code>terraform state mv</code>) | CLI command to update resource addresses in the state file without recreating resources. |

| | |
|-----------------------|--|
| Moved Block | A Terraform configuration block specifying <code>from</code> and <code>to</code> addresses to rename/move resources within the state during <code>plan</code> / <code>apply</code> . |
| Count Indexing | Assigning multiple instances of a resource by numeric index (e.g., <code>aws_instance.example[0]</code>). |
| For_Each Keys | Assigning multiple instances using a map/set of strings, each becoming a unique key. |
| Modules | Reusable Terraform configurations that group multiple resources under a logical unit. |

3. Why State Manipulation Is Needed

- Terraform identifies resources using **resource type + label**.
- Changing a label **without informing Terraform** looks like:
 - Old resource is **destroyed**
 - New resource is **created**
- This can cause **downtime, unnecessary costs, and risk**.

Example:

```

hcl
CopyEdit
# Original
resource "aws_instance" "old_label" { ... }

# Changed label
resource "aws_instance" "new_label" { ... }

```

Without moving state, Terraform sees this as:

```

diff
CopyEdit
- destroy aws_instance.old_label
+ create aws_instance.new_label

```

4. CLI Method – `terraform state mv`

Syntax:

bash

CopyEdit

```
terraform state mv [options] SOURCE DEST
```

- **SOURCE** = old resource address
- **DEST** = new resource address
- **Options:**
 - `-dry-run` → Shows what would be moved without changing state

Example – Simple Rename:

bash

CopyEdit

```
terraform state mv aws_instance.old_label aws_instance.new_label
```

 Updates state so Terraform knows it's the same resource

With Count Index:

bash

CopyEdit

```
terraform state mv 'aws_instance.old_list[0]'  
'aws_instance.new_list[0]'
```

Single quotes are needed because of `[]` (avoid shell interpretation issues)

With For_Each:

bash

CopyEdit

```
terraform state mv \  
  'aws_instance.old_list["instance1"]' \  
  'aws_instance.new_list["instance1"]'
```

5. Handling Resource Multiplication (Count & For_Each)

Count Example

```
hcl
CopyEdit
resource "aws_instance" "example" {
  count      = 2
  ami        = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

Terraform automatically maps:

```
arduino
CopyEdit
aws_instance.example[0]  # first instance
aws_instance.example[1]  # second instance
```

If increasing from 1 → 2 instances, Terraform will often **auto-move** the old resource to index [0].

For_Each Example

```
hcl
CopyEdit
locals {
  ec2_names = toset(["instance1", "instance2"])
}

resource "aws_instance" "example" {
  for_each      = local.ec2_names
  ami          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

Keys:

```
css
CopyEdit
```

```
aws_instance.example["instance1"]
aws_instance.example["instance2"]
```

CLI move example:

```
bash
CopyEdit
terraform state mv \
'aws_instance.old_list[0]' \
'aws_instance.example["instance1"]'
```

6. Configuration Method – Moved Blocks

Syntax:

```
hcl
CopyEdit
moved {
  from = aws_instance.old_name
  to   = aws_instance.new_name
}
```

Key Points:

- Declared in **Terraform configuration**, not CLI
- Preserves a **history** of refactors in code
- Executed during `terraform plan` / `terraform apply`
- Recommended for:
 - Documenting important moves
 - Ensuring future `apply` knows about past renames
- Avoid keeping hundreds of old moved blocks → adds noise

Example:

```
hcl
CopyEdit
```

```
hcl
CopyEdit
moved {
  from = aws_instance.new_list[0]
  to   = aws_instance.new_list["instance1"]
}
```

7. Moving Resources Into/Out of Modules

When moving into a module:

```
hcl
CopyEdit
moved {
  from = aws_instance.old_label
  to   = module.compute.aws_instance.this
}
```

When moving out of a module, reverse the addresses.

Address Pattern for Modules:

```
php-template
CopyEdit
module.<module_name>.<resource_type>.<resource_name>[<index/key>]
```

8. Full Example – CLI vs Moved Block

Original:

```
hcl
CopyEdit
resource "aws_instance" "old_label" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

Refactored with Moved Block:

```
hcl
CopyEdit
moved {
```

```

from = aws_instance.old_label
to   = aws_instance.new_label
}

resource "aws_instance" "new_label" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}

```

9. Step-by-Step Flow from Lecture

1. Initial Setup

- Create directory `16-state-manipulation`

Add `provider.tf` with:

```

hcl
CopyEdit
terraform {
  required_version = ">= 1.7"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 5.0"
    }
  }
}

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

```

-

Create EC2 Resource

```

hcl
CopyEdit
data "aws_ami" "ubuntu" {
  most_recent = true

```

```

owners      = ["099720109477"]
filter {
  name    = "name"
  values =
["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
}
}

resource "aws_instance" "old_label" {
ami          = data.aws_ami.ubuntu.id
instance_type = "t2.micro"
}

```

2.

Init & Apply

```

bash
CopyEdit
terraform fmt -recursive
terraform init
terraform apply

```

3.

4. Rename Resource

- Change label in code → causes Terraform to want to destroy & recreate

Prevent with:

```

bash
CopyEdit
terraform state mv aws_instance.old_label aws_instance.new_label

```

○

5. Expand with Count

- Terraform can auto-map index [0]

If label changes, move manually:

```

bash
CopyEdit

```

```
terraform state mv 'aws_instance.old_label[0]'  
'aws_instance.new_label[0]'
```

○

6. Switch to For_Each

Move using keys:

```
bash  
CopyEdit  
terraform state mv \  
'aws_instance.old_list[0]' \  
'aws_instance.new_list["instance1"]'
```

○

Refactor with Moved Block

```
hcl  
CopyEdit  
moved {  
  from = aws_instance.new_list[0]  
  to   = aws_instance.new_list["instance1"]  
}
```

7.

Move to Module

```
hcl  
CopyEdit  
moved {  
  from = aws_instance.new_list["instance2"]  
  to   = module.compute.aws_instance.this  
}
```

8.

10. Best Practices

- Use **moved blocks** for documented, repeatable refactors

- Keep only **relevant** moved blocks (remove ancient noise)
 - Use single quotes ' in CLI when addresses contain brackets
 - Always `terraform plan` before moving to avoid mistakes
 - When moving into modules, know exact resource addresses inside module
-

11. Common Pitfalls

- ✗ Forgetting `terraform init` after adding a module
- ✗ Not quoting addresses with [] in CLI → shell misinterprets
- ✗ Moving wrong resource → Terraform may lose mapping & recreate
- ✗ Over-keeping moved blocks → bloats config

Lecture 3 — Importing Existing Infrastructure into Terraform

1. What is Terraform Import?

Terraform Import lets you bring **already-existing resources** (created manually or by another tool) under Terraform's control **without recreating them**.

- Common use cases:
 - Migrating manually created AWS resources into Terraform.
 - Transitioning resources between teams/projects.
 - Taking ownership of resources created by other IaC tools.

Important:

Import only **adds the resource to the Terraform state**; it does **not** generate the `.tf` code automatically. You must still write the resource definition yourself.

2. Key Concepts

| Term | Meaning |
|----------------------|--|
| State file | Terraform's record of infrastructure, mapping <code>.tf</code> code to real-world resources. |
| Import target | The resource address in Terraform syntax (e.g., <code>aws_s3_bucket.remote_state</code>). |
| Resource ID | The unique ID of the real resource in the provider (e.g., bucket name for S3). |
| Import block | Terraform code block that imports resources declaratively. |
| CLI import | Command-line method to import existing resources into state. |

3. Import Methods

Terraform supports **two import approaches**:

A. CLI Method

bash

CopyEdit

```
terraform import <resource_address> <resource_id>
```

- `resource_address`: `resource_type.resource_name` (from `.tf` file).
- `resource_id`: Identifier of the existing resource (depends on provider).

Example: Import S3 Bucket via CLI

```
bash
CopyEdit
terraform import aws_s3_bucket.remote_state
my-terraform-remote-backend
```

B. Import Block Method

```
hcl
CopyEdit
import {
  to = aws_s3_bucket.remote_state
  id = "my-terraform-remote-backend"
}
```

- Placed **inside `.tf` files**.
 - Runs with `terraform apply`.
 - Can be kept for documentation/history.
-

4. Steps to Import an Existing Resource

Step 1: Write the Resource Block

Terraform needs a matching resource block before import.

Example (`import.tf`):

```
hcl
CopyEdit
resource "aws_s3_bucket" "remote_state" {
  bucket = "my-terraform-remote-backend"
```

```
lifecycle {  
    prevent_destroy = true  
}  
}
```

Step 2: Identify the Resource ID

Check provider documentation → **Import section**.

For **AWS S3 Bucket**, ID = bucket name.

Step 3: Import

- **CLI Method**

```
bash  
CopyEdit  
terraform import aws_s3_bucket.remote_state  
my-terraform-remote-backend
```

- **Import Block Method**

```
hcl  
CopyEdit  
import {  
    to = aws_s3_bucket.remote_state  
    id = "my-terraform-remote-backend"  
}
```

Run:

```
bash  
CopyEdit  
terraform apply
```

Step 4: Verify

- Check state file:

```
bash
CopyEdit
terraform state list
terraform state show aws_s3_bucket.remote_state
```

- Confirm Terraform now tracks the resource.
-

5. Detailed Example — S3 Bucket Import

We created an **S3 bucket manually** for remote backend storage, but now we want Terraform to manage it.

Resource Definition (`import.tf`):

```
h
CopyEdit
resource "aws_s3_bucket" "remote_state" {
  bucket = "my-terraform-remote-backend"

  lifecycle {
    prevent_destroy = true
  }

  tags = {
    ManagedBy = "Terraform"
    Lifecycle = "Critical"
  }
}
```

CLI Import Command:

```
bash
CopyEdit
terraform import aws_s3_bucket.remote_state
my-terraform-remote-backend
```

What happens:

- Terraform updates the **state file** with real-world resource attributes.
 - No infrastructure changes yet — Terraform just starts tracking.
-

6. Edge Cases & Behavior

1. Different Name in Code vs. Resource

- Terraform imports actual live configuration into state.
- If `.tf` file doesn't match → next `terraform plan` will show changes (possibly replacement).

2. Dry Run Limitations

- **CLI import** → no `--dry-run`.
- **Import block** → can dry-run via `terraform plan` before applying.

Prevent Accidental Deletion

Use:

```
hcl
CopyEdit
lifecycle {
  prevent_destroy = true
}
```

- 3. Useful for critical resources like remote state buckets.
-

7. Importing Related Resources

Example: S3 Bucket Public Access Block

If bucket already has public access settings, we can import that too.

Resource:

```
hcl
CopyEdit
resource "aws_s3_bucket_public_access_block" "remote_state" {
  bucket          = aws_s3_bucket.remote_state.bucket
```

```
    block_public_acls      = true
    block_public_policy    = true
    ignore_public_acls    = true
    restrict_public_buckets = true
}
```

Import Block:

```
h
CopyEdit
import {
  to = aws_s3_bucket_public_access_block.remote_state
  id = aws_s3_bucket.remote_state.bucket
}
```

Run:

```
bash
CopyEdit
terraform apply
```

8. CLI vs Import Block — Pros & Cons

| Method | Pros | Cons |
|--------------|--|--|
| CLI | Quick one-time import | No record in code; harder to track later |
| Import Block | Persistent record in <code>.tf</code> ; supports dry-run | Slightly slower workflow |

9. Best Practices

- Always check **provider docs** for correct import ID format.
- Keep **imports** in a dedicated `imports.tf` file for history.
- Add **tags** and `prevent_destroy` for critical resources.
- After import, run `terraform plan` to sync configurations.

10. Verification Commands

```
bash
CopyEdit
terraform state list    # List all resources in state
terraform state show aws_s3_bucket.remote_state  # Show specific
resource details
terraform plan          # See what Terraform will change
terraform apply         # Apply changes
```

11. Code Recap

Full Working Example (`import.tf`):

```
hcl
CopyEdit
resource "aws_s3_bucket" "remote_state" {
  bucket = "my-terraform-remote-backend"

  lifecycle {
    prevent_destroy = true
  }

  tags = {
    ManagedBy = "Terraform"
    Lifecycle = "Critical"
  }
}

resource "aws_s3_bucket_public_access_block" "remote_state" {
  bucket              = aws_s3_bucket.remote_state.bucket
  block_public_acls   = true
  block_public_policy = true
  ignore_public_acls = true
  restrict_public_buckets = true
}

import {
  to = aws_s3_bucket_public_access_block.remote_state
  id = aws_s3_bucket.remote_state.bucket
}
```

}

✓ **In short:** Import only changes Terraform's **state file** — not the infrastructure. You must still write `.tf` definitions, and you should ensure they match the real-world configuration to avoid Terraform replacing resources unexpectedly.

Lecture 4 — Removing Infrastructure from Terraform Without Destroying It

Core Concept

In Terraform, every resource you define in code is **tracked in Terraform State**.

If you delete the resource block from `.tf` files and run `terraform apply`, Terraform will **destroy** that resource in real life.

But sometimes you want to **stop managing** a resource **without deleting it** from your cloud provider.

Why Would You Do This?

Use cases:

1. **Handing resources over to another team**

Example: A shared S3 bucket now maintained by the storage team instead of you.

2. **Splitting a monolithic Terraform project into multiple smaller ones**

You want to move the resource to another state file.

3. **Migrating to another IaC tool**

You want to stop Terraform tracking it, but keep it alive.

4. **Manual management**

Resource is now handled manually or via AWS Console.

What Happens If You Just Comment Out/Delete the Resource Block?

- Terraform sees the resource **exists in state but not in code**.
 - **Terraform plan** → proposes **destroy** operation.
 - If you `apply`, Terraform deletes the actual cloud resource.
-

Solution: Stop Tracking Without Destroying

Terraform allows two ways to **unlink a resource from state**:

Method 1 — CLI Command

Command:

```
bash
CopyEdit
terraform state rm <resource_address>
```

Example:

```
bash
CopyEdit
terraform state rm aws_s3_bucket.my_bucket
```

Optional Dry Run:

```
bash
CopyEdit
terraform state rm -dry-run aws_s3_bucket.my_bucket
```

Shows what will be removed from state without actually doing it.

Effect:

- Removes the resource entry from the state file.
- Terraform will “forget” it.
- Resource remains untouched in AWS.

Steps:

1. Create or identify the resource in Terraform (example: `aws_s3_bucket.my_bucket`).
2. Run `terraform state rm aws_s3_bucket.my_bucket`.

3. Delete the `.tf` block for that resource.
 4. Run `terraform apply` → no changes planned.
-

Method 2 — `removed` Block (Terraform v1.7+)

The `removed` block is a new Terraform feature for **documented removal**.

Syntax:

```
hcl
CopyEdit
removed {
  from = aws_s3_bucket.my_bucket
  lifecycle {
    destroy = false
  }
}
```

Arguments:

- `from` — Resource address in state (required).
 - `lifecycle.destroy` —
 - `false` → Just remove from state (default safe option).
 - `true` → Also destroy the real resource.
-

Examples

Keep Resource in AWS, Stop Tracking

```
hcl
CopyEdit
removed {
  from = aws_s3_bucket.my_bucket
  lifecycle {
    destroy = false
  }
}
```

```
    }
}
```

- Resource stays in AWS.
 - Terraform forgets it.
-

Destroy Resource While Removing from Code

```
hcl
CopyEdit
removed {
  from = aws_s3_bucket.my_bucket
  lifecycle {
    destroy = true
  }
}
```

- Resource deleted from AWS **and** from state.
-

Why Use **removed** Block Over CLI?

- **Documentation benefit:** The removal is visible in code & version control.
 - No need to remember CLI commands for later.
 - Works well when multiple people manage the project.
-

Full Workflow Example

1 Create Resource

```
remove.tf

hcl
CopyEdit
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-random-bucket-1234"
```

```
}
```

```
bash
CopyEdit
terraform apply
```

✓ Bucket created in AWS & tracked in state.

2 Decide to Stop Tracking

If you just delete the block:

```
bash
CopyEdit
terraform apply
```

✗ Terraform will destroy the bucket.

3 Remove via CLI (Safe Way)

```
bash
CopyEdit
terraform state rm aws_s3_bucket.my_bucket
```

Then delete resource block from .tf file.

4 Or Remove via removed Block

```
hcl
CopyEdit
removed {
  from = aws_s3_bucket.my_bucket
  lifecycle {
    destroy = false
  }
}
```

Run:

```
bash
CopyEdit
terraform apply
```

- ✓ State forgets resource, AWS keeps it.
-

Important Notes

- **Import after removal:** If you want to manage the resource again later, use:

```
bash
CopyEdit
terraform import aws_s3_bucket.my_bucket my-bucket-name
```

- **removed** block **does not** preserve full resource configuration — only the address.
 - For older Terraform (<1.7), **removed** block **does not exist** → you must use CLI.
-

Cheat Sheet

| Action | Resource Exists in AWS? | Resource in State? | Command/Block |
|--------------------------------------|-------------------------|--------------------|--|
| Stop tracking only | ✓ | ✗ | <code>terraform state rm</code> or
<code>removed { destroy = false }</code> |
| Destroy resource & remove from state | ✗ | ✗ | <code>removed { destroy = true }</code> or delete block & apply |
| Import existing | ✓ | ✓ | <code>terraform import</code> |
| Just delete from config | ✗ | ✗ | Delete block & apply (destroys) |

Code Summary

hcl

```
CopyEdit
# Example bucket
resource "aws_s3_bucket" "my_bucket" {
    bucket = "my-bucket-12345"
}

# Removing safely
removed {
    from = aws_s3_bucket.my_bucket
    lifecycle {
        destroy = false # change to true to delete
    }
}

bash
CopyEdit
# CLI alternative
terraform state rm aws_s3_bucket.my_bucket
```

If you want, I can also make you a **full diagram showing how state changes in both CLI and removed-block methods** so you visually remember it forever.
Do you want me to make that?

Lecture 5 — Working with Tainted / Unchanged Resources in Terraform

1. Key Concept: Why We Need to Replace Resources Without Config Changes

- Sometimes a resource in AWS (or other providers) gets into an **invalid state** — misconfigured, corrupted, or out-of-sync — *even though Terraform configuration hasn't changed*.
- Normally, Terraform won't recreate a resource unless config changes.
- **Solution:** Force Terraform to replace it manually, without editing the `.tf` file.

Examples in real-world:

- **S3 Bucket:** Permissions accidentally changed in AWS Console.
- **Lambda Function URL:** Becomes disconnected from the function after AWS backend changes.
- **VPC/Subnet IDs:** Change unexpectedly, breaking dependencies.

2. Core Terraform Concepts in This Lecture

| Term | Meaning | Real-world Analogy |
|-----------------------|---|---|
| Taint | Mark a resource as "bad" so Terraform will destroy and recreate it on next <code>apply</code> . | Slapping a "Defective — Replace" sticker on a machine. |
| Untaint | Remove the taint mark so Terraform won't replace the resource. | Taking off the sticker before sending it to repair. |
| Replace Flag | Newer, recommended way to achieve taint effect: <code>terraform apply -replace=...</code> | Skipping the sticker and directly ordering a replacement in one step. |
| Cascading Replacement | When replacing a resource forces replacements of dependent resources. | Replacing a motherboard means replacing CPU & RAM too. |

| | | |
|-----------------------------|--|---|
| Dependency Awareness | Terraform decides whether to replace downstream resources based on changed properties. | Changing a room's paint doesn't require replacing the furniture unless it's attached to the wall. |
|-----------------------------|--|---|

3. Example 1: Simple S3 Bucket Recreation

File: `taint.tf`

```
hcl
CopyEdit
resource "aws_s3_bucket" "tainted" {
  bucket = "my-tainted-bucket-1234"
}
```

Workflow:

Create bucket:

```
bash
CopyEdit
terraform apply
```

1.

Force replace:

```
bash
CopyEdit
terraform taint aws_s3_bucket.tainted
terraform apply
```

2.

Cancel replacement:

```
bash
CopyEdit
terraform untaint aws_s3_bucket.tainted
```

3.

4. Example 2: Dependency Replacement (VPC → Subnet)

```
h  
CopyEdit  
resource "aws_vpc" "this" {  
    cidr_block = "10.0.0.0/16"  
}  
  
resource "aws_subnet" "this" {  
    cidr_block = "10.0.0.0/24"  
    vpc_id     = aws_vpc.this.id  
}
```

Scenario:

Taint VPC:

```
bash  
CopyEdit  
terraform taint aws_vpc.this  
terraform apply
```

-
- Result: **VPC ID changes** → **Subnet replaced** because its `vpc_id` property changes.

Important: Only properties **referenced** in other resources trigger cascading replacements.

5. Example 3: When Dependencies Don't Get Replaced

```
hcl  
CopyEdit  
resource "aws_s3_bucket" "tainted" {  
    bucket = "example-tainted-bucket"  
}  
  
resource "aws_s3_bucket_public_access_block" "from_tainted" {  
    bucket           = aws_s3_bucket.tainted.id  
    block_public_acls = true  
    block_public_policy = true
```

}

- If we **taint** the bucket, Terraform **may not** recreate the `public_access_block` if **bucket name** hasn't changed.
- This can lead to **inconsistent state**:
 - Bucket replaced
 - Public access block still attached to the *old* bucket's settings.

Solution: Explicitly replace both:

```
bash
CopyEdit
terraform apply -replace=aws_s3_bucket.tainted
-replace=aws_s3_bucket_public_access_block.from_tainted
```

6. `terraform taint` vs `terraform apply -replace`

| Feature | <code>terraform taint</code> | <code>terraform apply -replace</code> |
|--------------|---------------------------------------|---------------------------------------|
| Availability | Older command (considered deprecated) | Current recommended method |
| Steps | 2-step: taint → apply | 1-step apply |
| Scope | Marks in state file for next run | Forces replacement during same run |
| Risk | Easy to forget to apply afterward | More direct, less error-prone |

7. Best Practices

1. **Check dependencies before replacement** — Avoid partial replacements causing broken links.
2. **Use `-replace` for precision** — Especially in production.
3. **Run `terraform plan` before apply** — Always verify cascade effects.

4. **Be aware of lifecycle rules** — `prevent_destroy` can block deletion.
 5. **Clean up empty S3 buckets manually** before destroy — AWS won't delete non-empty buckets.
-

8. Lifecycle `prevent_destroy` Reminder

```
hcl
CopyEdit
lifecycle {
  prevent_destroy = true
}
```

- Blocks `terraform destroy` and replacements.
 - Use on **critical resources** like remote state buckets.
 - Must be removed or overridden to delete.
-

9. Commands Cheat Sheet

```
bash
CopyEdit
# Mark a resource as tainted
terraform taint aws_s3_bucket.tainted

# Remove taint
terraform untaint aws_s3_bucket.tainted

# Replace using new method
terraform apply -replace=aws_s3_bucket.tainted

# Replace multiple resources
terraform apply \
  -replace=aws_s3_bucket.tainted \
  -replace=aws_s3_bucket_public_access_block.from_tainted

# Plan without changes
```

```
terraform plan  
  
# Destroy everything  
terraform destroy
```

10. Real-World Debugging Flow

1. **Symptom:** AWS Console shows a resource misbehaving (permissions wrong, ID mismatch).
2. **Check Terraform config** — If unchanged, Terraform won't touch it.

Force replace:

```
bash  
CopyEdit  
terraform apply -replace=resource_type.resource_name
```

- 3.
 4. **Check dependencies** — Replace dependent resources if needed.
 5. **Verify in Console** after apply.
-

11. Summary Mindmap

- **When?**
 - Resource in bad state
 - No config changes detected
- **How?**
 - Old way: `terraform taint` + `terraform apply`
 - New way: `terraform apply -replace=...`
- **Risks:**

- Partial replacements breaking dependencies
 - AWS limitations (e.g., non-empty S3 deletion)
- **Prevention:**
 - Plan first
 - Replace dependencies explicitly
-

If you want, I can also make you a **one-page visual cheat sheet** with **commands, flowcharts, and common pitfalls** for tainting/replacing in Terraform — so you can keep it as a reference when working in AWS.

Do you want me to prepare that next?