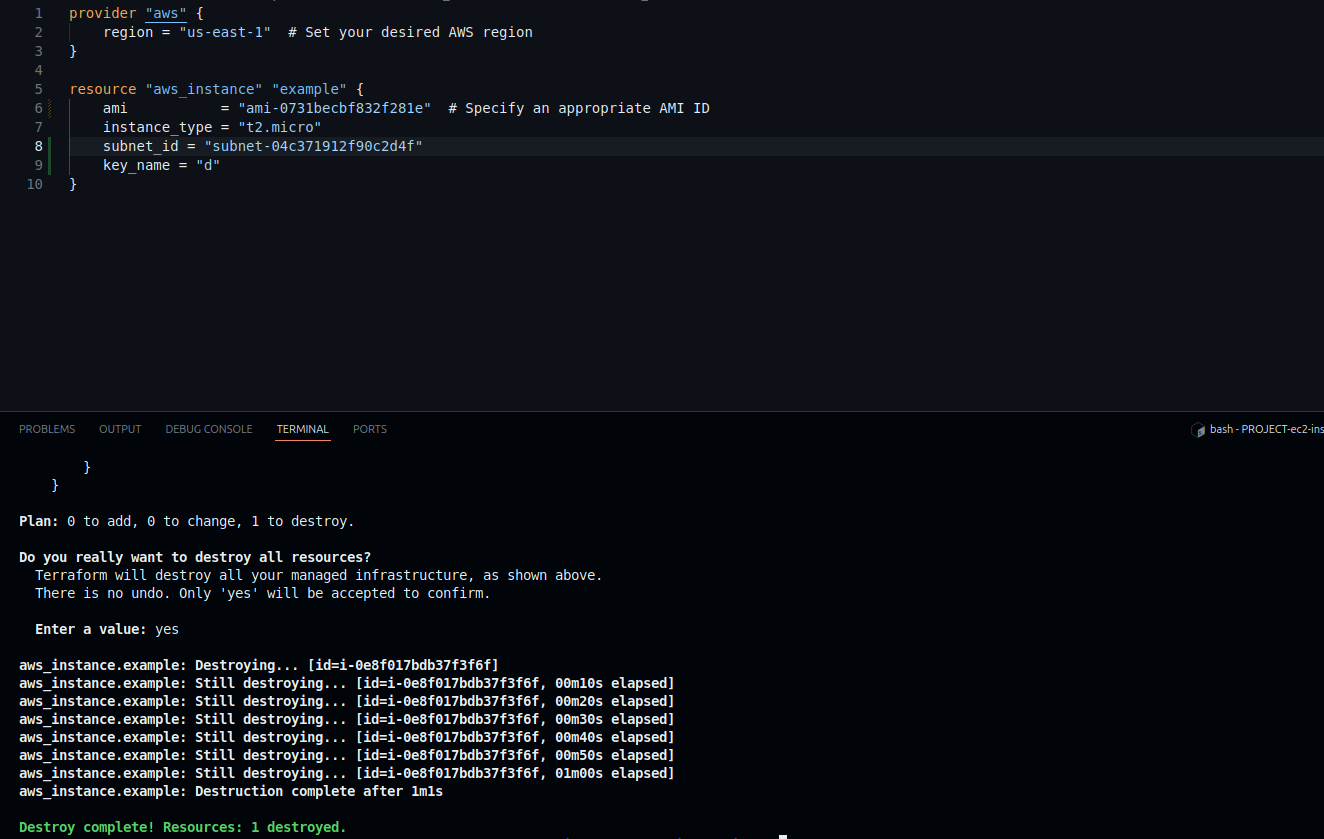
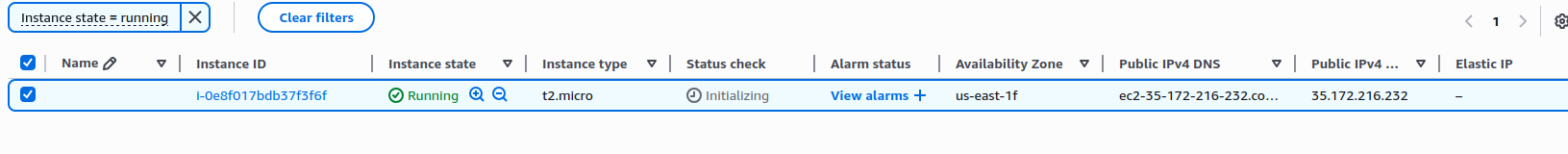
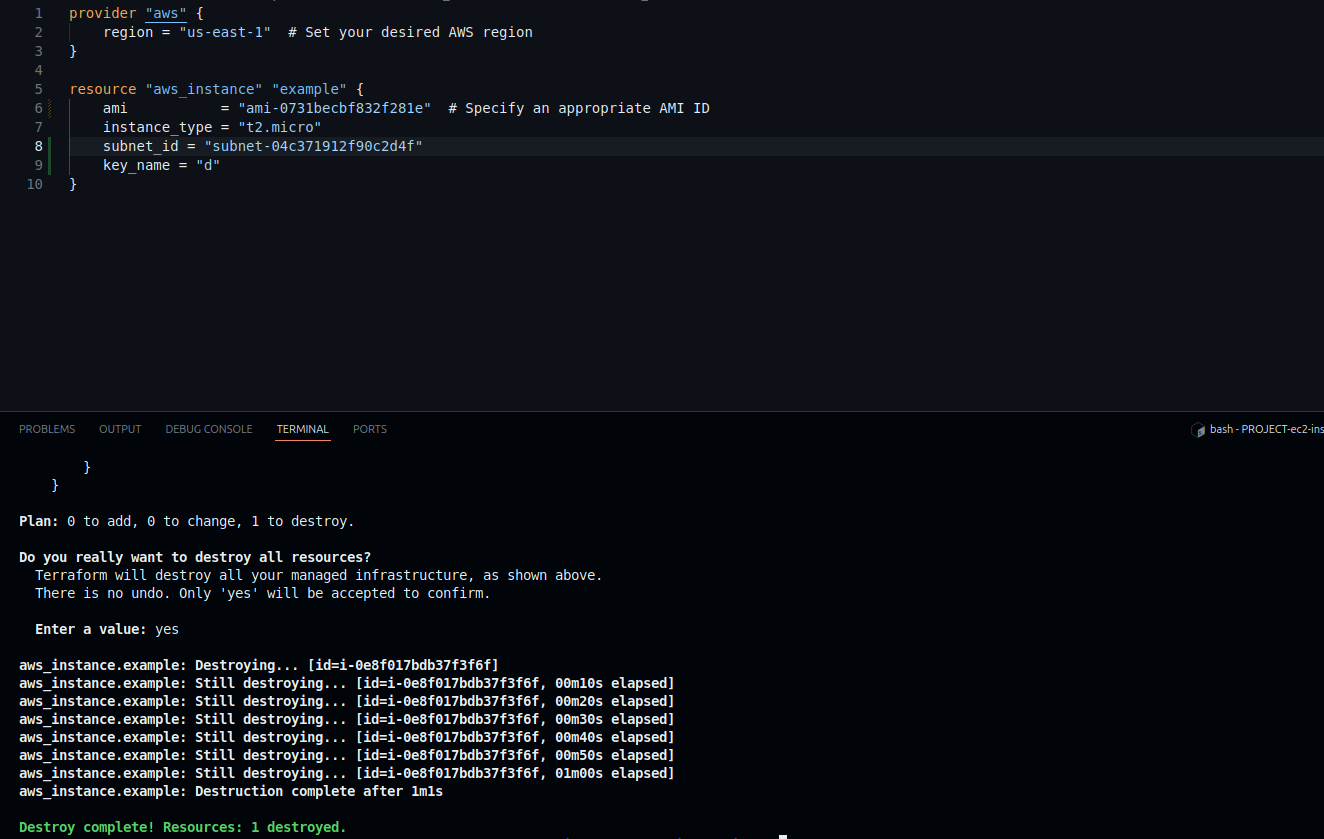
Name –Aasif mohd  
Project – Terraform

I wont’t add the installation part it is pretty easy can be found on youtube  
Task 1. Create an EC2 instance using terraform on AWS.

Provider:- A provider is a plugin that helps Terraform to understand where it has to create the infrastructure  
  
Solution.  
1.First setup the provider aws  
2. write the code for instance-type,id,key etc.

3. Check the implementation using terraform plan cammand

4. use the cammand terraform apply to create an instance  
instance is created on aws

Now to delete this instance we can use the cammand-terraform destroy  
Task 2----  
What is a variable in terraform?

A **variable** in Terraform is like a placeholder. It lets you **store a value** (like a region name, server type, etc.) so you can **reuse** it in your code.

Eg. ----  
variable "region" {

default = "us-east-1"

}

### What is an Output ?

An **output or output variable** shows **useful information** after you run terraform apply. It's like Terraform saying:  
 **“Here’s the result you might want to see!”**

### Why Use Output ?

* To see values like IP addresses, bucket names, or resource IDs.
* To pass values between modules.
* To debug and check values.

output "bucket\_name" {

value = aws\_s3\_bucket.my\_bucket.bucket

}  
  
  
🌥️ **Multicloud in Terraform** (Multiple Cloud Providers)

* **What it means:** Using **more than one cloud provider** in the same Terraform project.
* **Example:** You deploy a web app on **AWS** and a database on **Azure**.
* **Why it's used:** To avoid putting all your services on one provider (vendor lock-in), or to take advantage of features from different providers.

**In Terraform:**  
 You use **multiple provider blocks**.  
Example-

provider "aws" {

region = "us-east-1"

}

provider "azurerm" {

features = {} }

### **Multiregion in Terraform** (Multiple Regions in One Cloud)

* **What it means:** Deploying resources in **multiple regions** of the **same cloud provider** (like AWS).
* **Example:** Deploying servers in **us-east-1** and **us-west-1** to improve performance or reliability.
* **Why it's used:** For disaster recovery, performance, or compliance.

**In Terraform:**  
 You use **multiple provider blocks** for the same cloud but with different regions.

Example-

provider "aws" {

alias = "us\_east"

region = "us-east-1"

}

provider "aws" {

alias = "us\_west"

region = "us-west-1"

}

resource "aws\_instance" "east\_server" {

provider = aws.us\_east

# ...

}

resource "aws\_instance" "west\_server" {

provider = aws.us\_west

# ...

}  
  
  
Module  
What is a Module in Terraform?

A **module** in Terraform is a **folder** that contains Terraform code and can be **reused** in different places by passing values to it.

A **module** is just a **folder** that contains Terraform code (like resources, variables, outputs) — it’s like a **reusable component**.

Think of it like a **Lego block**:

* You build it once.
* You can plug it into multiple places.
* You can pass values to it, and it gives outputs back.

### Why use Modules?

* To **organize** your code.
* To **reuse** the same setup (e.g., EC2 instance, VPC) in multiple environments.
* To **avoid copy-pasting** the same code everywhere.

**Value from main.tf** → goes into → **variable block in module** → used in → **resource or logic inside module's main.tf**

We can use the module code from the outside of module folder easily without have to write the complex code

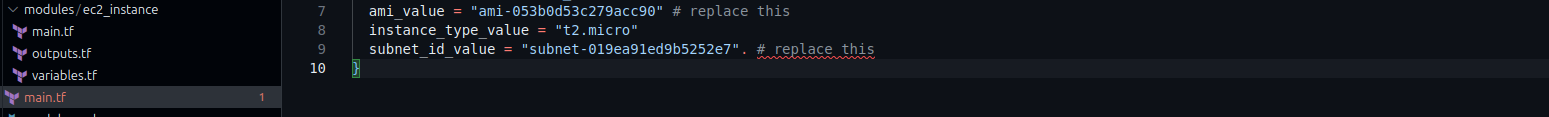
modules/

└── ec2\_instance/

├── main.tf # Contains resources

├── variables.tf # Defines input variables

└── outputs.tf # Defines output values



### What is a Terraform State File?

When you use **Terraform** to create cloud resources (like servers or databases), it needs to **remember what it created**. That's what the **state file** does!

* It's a file called terraform.tfstate.
* It keeps track of **all the resources** Terraform manages.
* Without it, Terraform wouldn't know what exists or what needs to be changed.

#### 1. **Security Risk (Sensitive Data)**

* The state file may contain **secrets** like passwords, API keys, or cloud resource IDs.
* If pushed to GitHub, even in private repos, there's a risk of **accidental leaks**.-

--A developer might **clone** the repo to another machine or **copy/paste** it elsewhere (like in a public issue, gist, or Slack).

#### 2. **No Locking or Sync**

* GitHub doesn't support **locking** — if two people push or pull at the same time, it can cause **conflicts or corrupted state**.
* It's not designed for **real-time collaboration** on infrastructure.

### How S3 Fixes These Issues

#### 1. **Secure & Encrypted Storage**

* S3 allows you to enable **server-side encryption**.
* You can set **IAM permissions** to tightly control who can access the file.

#### 2. **Supports Locking & Versioning**

* With **S3 + DynamoDB**, Terraform can **lock the state** so only one person can change it at a time.
* **Versioning** in S3 means you can roll back to older state files if something breaks.

### Workflow: Using GitHub + S3 for Terraform

1. **Terraform Code in GitHub**
   1. The DevOps engineer pulls only the **Terraform code** (like .tf files) from GitHub.
   2. The **state file is not in GitHub** — it’s safely stored in **S3**.
2. **Run terraform init**
   1. This connects Terraform to the **remote backend (S3)** using the config in backend "s3" block.
   2. Terraform now knows where the state lives.
3. **Make Changes & Run terraform apply**
   1. Terraform compares the **current state in S3** with your updated code.
   2. It updates infrastructure and automatically **writes the new state** back to S3.
4. **State File is Updated in S3 Automatically**
   1. You don’t need to download or upload the state manually.
   2. S3 handles storage, and **DynamoDB (optional)** handles locking.
5. **If You Make a Mistake? No Worries!**
   1. If something breaks, you can **restore a previous version** of the state file from S3’s **versioning** feature.
   2. That makes it easy to recover from mistakes.

**if you're not using remote state** (like S3) and someone forgets to **push or update the state file**, there’s a **real risk of duplicate resources being created.**

Let’s break it down:

### When Using Local State and GitHub Only (Bad Practice)

1. **Engineer A**
   1. Clones the repo.
   2. Applies Terraform with **local state file**.
   3. Creates, for example, an EC2 instance.
   4. **Forgets to push the updated state file** to GitHub (or it’s not in GitHub at all).
2. **Engineer B**
   1. Pulls the same repo, **but without the latest state**.
   2. Terraform sees **no record** of the EC2 instance.
   3. So when B runs terraform apply, it **creates another EC2 instance** — a **duplicate**.

### What does "provision" mean in Terraform?

In **Terraform**, **provisioning** means **setting up or configuring** a resource **after** it's created. It’s like saying:

“Hey Terraform, after you create this server, run this script to install software or make some changes inside it.”

### Example to understand it:

Imagine you're using Terraform to create a **virtual machine (VM)** in the cloud. That’s easy:

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

}

This creates a VM. But what if you want that VM to:

* Install **Nginx**?
* Set up **some files**?
* Run a **shell script**?

That’s where **provisioners** come in.

### 🛠 Example with a provisioner:

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "remote-exec" {

inline = [

"sudo apt update",

"sudo apt install -y nginx"

]

}

}

This tells Terraform:

“After you create the VM, connect to it and run these commands.”

### Types of Provisioners:

1. **file** – Uploads a file to the server
2. **remote-exec** – Runs commands on the server (SSH)
3. **local-exec** – Runs a command on your local machine (not the server)

### 🔹 Terraform Workspaces (Quick Explanation)

**Terraform workspaces** let you use the **same code** for different environments (like dev, stage, prod) by creating a **separate state file** for each one.

So:

* One codebase ✅
* Multiple environments ✅
* Separate .tfstate files ✅

### Why Use Workspaces?

You might have one Terraform codebase (like for setting up a server), but want:

* One for **development** (dev)
* One for **production** (prod)

Instead of copying the code multiple times, you can:

* Keep **one codebase**
* Use **different workspaces** for each environment

Each workspace has its **own separate state**.

|  |  |
| --- | --- |
| **Command** | **What It Does** |
| terraform workspace list | Shows all available workspaces |
| terraform workspace new dev | Creates a new workspace named dev |
| terraform workspace select dev | Switches to the dev workspace |
| terraform workspace show | Shows the current workspace |

### Terraform Secrets with Vault

**Vault** is a tool to **securely store secrets** like passwords, API keys, and access tokens.

Instead of putting secrets directly in your Terraform code (which is risky), you can:

1. **Store secrets in Vault**
2. **Configure Terraform to read those secrets**
3. **Use them in your infrastructure setup**

### Step-by-Step Process

1. **Secrets Stored in Vault**
   1. Example: Store db\_password = "supersecret" in Vault at path like secret/data/db.
2. **Terraform Authenticates with Vault**
   1. Terraform uses an **AppRole**, **Token**, or other auth method.
   2. Vault checks if Terraform is allowed to access the secret.
3. **Terraform Reads Secret**
   1. Terraform uses the **Vault provider** to fetch the secret.
   2. The secret is used during provisioning (like setting up a database).
4. **No Secrets in Code!**
   1. The sensitive data stays in Vault.
   2. Terraform just reads it at runtime.

### Why Use Vault with Terraform?

* **Security**: No hardcoding secrets in code
* **Central management**: Keep all secrets in one place
* **Auditing**: See who accessed what and when