😊 Terraform Creating an EC2 Instance

Terraform is an open-source infrastructure as code software tool created by HashiCorp. It allows users to define and provision data center infrastructure using a declarative configuration language. With Terraform, you can manage various cloud services, on-premises infrastructure, and other external services, such as DNS providers or SaaS providers.

The key concept in Terraform is the Terraform configuration files, typically written in HashiCorp Configuration Language (HCL). These files describe the desired state of your infrastructure, including resources like virtual machines, networks, storage, and more. Terraform then compares this desired state with the current state of the infrastructure and makes the necessary changes to bring the infrastructure to the desired state.

Terraform provides a consistent workflow for managing infrastructure changes, including creating, updating, and destroying resources. It also supports state management, allowing you to store the state of your infrastructure in a central location, such as a remote backend, to enable collaboration and consistency among team members.

😄 Use cases of Terraform:

Terraform is versatile and can be used in various scenarios across different industries. Here are some common use cases:

1. **Infrastructure Provisioning:** Terraform is commonly used to provision and manage infrastructure resources on various cloud platforms such as AWS, Azure, Google Cloud Platform (GCP), and others. It allows you to define your infrastructure as code, enabling repeatable and consistent deployments.
2. **Multi-Cloud Deployment:** Organizations often use multiple cloud providers or a combination of cloud and on-premises resources. Terraform enables managing multi-cloud environments using a single configuration language, providing consistency and simplifying management across different platforms.
3. **Immutable Infrastructure:** Terraform supports the concept of immutable infrastructure, where infrastructure components are replaced rather than modified. This approach enhances reliability, security, and scalability by ensuring consistency and predictability in deployments.
4. **Continuous Integration/Continuous Deployment (CI/CD):** Terraform can be integrated into CI/CD pipelines to automate the deployment of infrastructure changes. This allows for faster delivery of updates and reduces the risk of errors associated with manual interventions.
5. **Disaster Recovery:** Terraform facilitates the creation of disaster recovery environments by defining backup infrastructure configurations. In the event of a disaster, these configurations can be quickly deployed to restore services and minimize downtime.
6. **DevOps Automation:** Terraform plays a crucial role in DevOps practices by automating the provisioning and management of infrastructure. It enables collaboration between

development and operations teams, accelerates the delivery of applications, and improves overall efficiency.

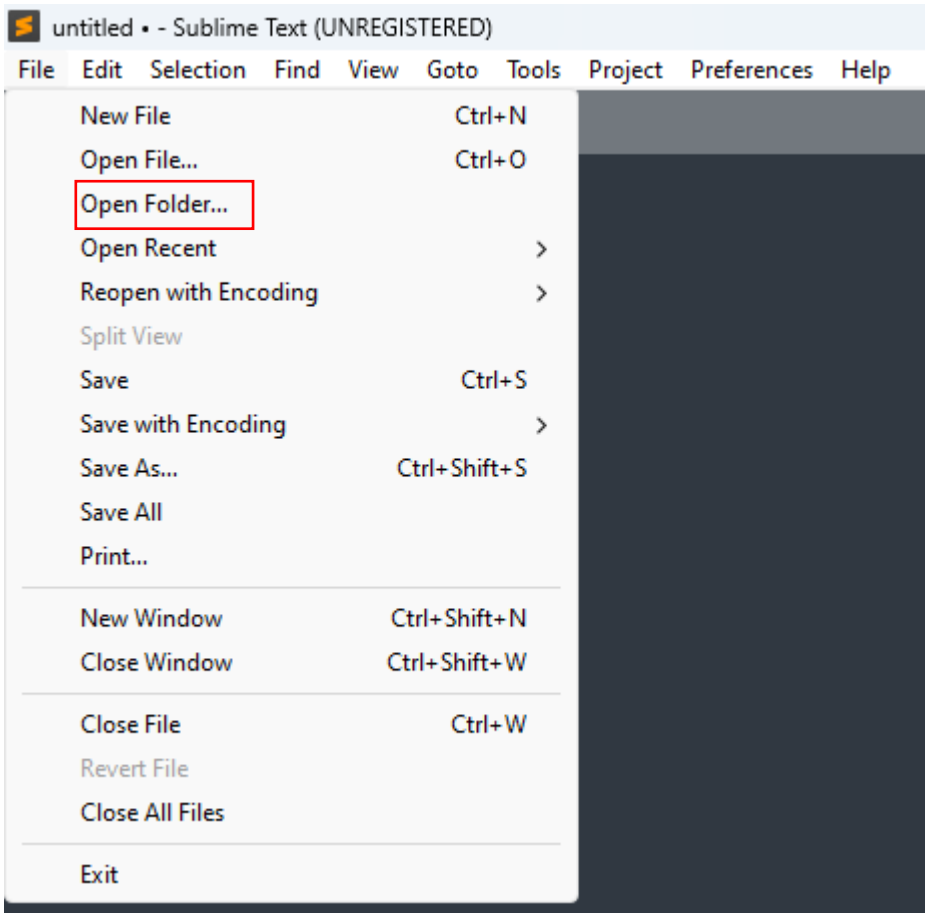
7. **Scaling Infrastructure:** As your application or business grows, you may need to scale your infrastructure to handle increased demand. Terraform allows you to dynamically scale resources up or down based on workload requirements, ensuring optimal performance and cost efficiency.
8. **Compliance as Code:** Terraform can be used to enforce compliance policies by codifying security and regulatory requirements into infrastructure configurations. This ensures that infrastructure deployments adhere to organizational standards and industry regulations.
9. **Testing Environments:** Terraform facilitates the creation of consistent testing environments, including staging and development environments, which closely resemble production. This enables teams to test changes in a controlled environment before deploying to production.
10. **Microservices Orchestration:** Terraform can be used to manage the infrastructure required for deploying and scaling microservices architectures. It enables the creation and management of container clusters, load balancers, service discovery, and other components essential for microservices deployments.

😁 To begin with the Lab:

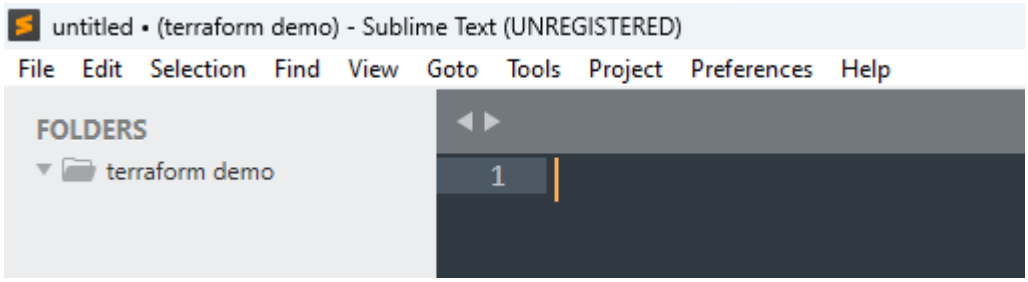
1. Now to use terraform first you need to install it in your local machine which might be your laptop or desktop.
2. For that search Download Terraform on google or if you have chocolaty in your system then you can simply run the command **choco install terraform.**
3. Remember to install terraform using chocolaty you need to open PowerShell as an administrator.
4. Now I would prefer to install Terraform using chocolaty.
5. Once you have setup your Terraform open GitBash and type **terraform --help**
6. You will see this much of information popping up.

```
$ terraform --help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  metadata      Metadata related commands
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Execute integration tests for Terraform modules
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

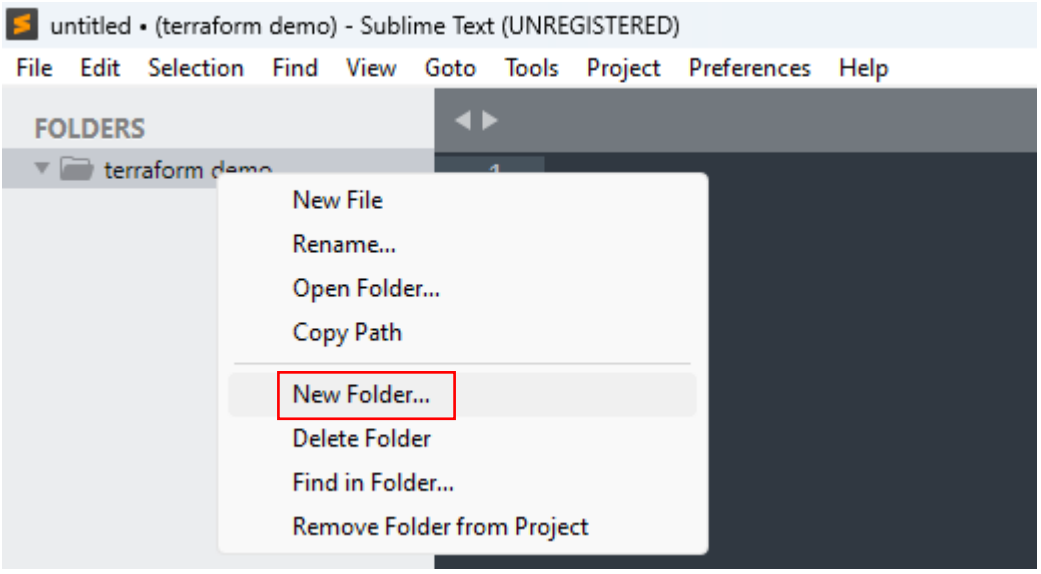
7. Now to use terraform you need to write a code. So, I've had that and you can get that code on GitHub.
8. Now I am using Sublime Text editor to write this code for terraform. You can use any code editor of your choice. One more thing you can install this code editor using chocolaty.
9. Now once you have set the environment its time to write the code. Open your code editor for me it is sublime text editor.
10. There in the code editor from the top left corner click on file and open an empty folder. If you don't have any then create one and open it then.



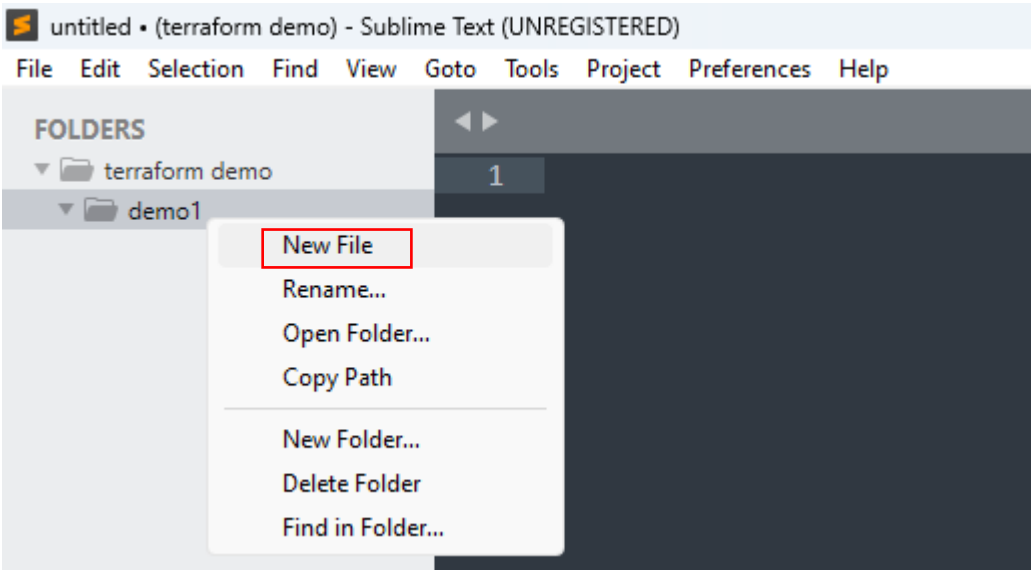
11. Now it would look like this.



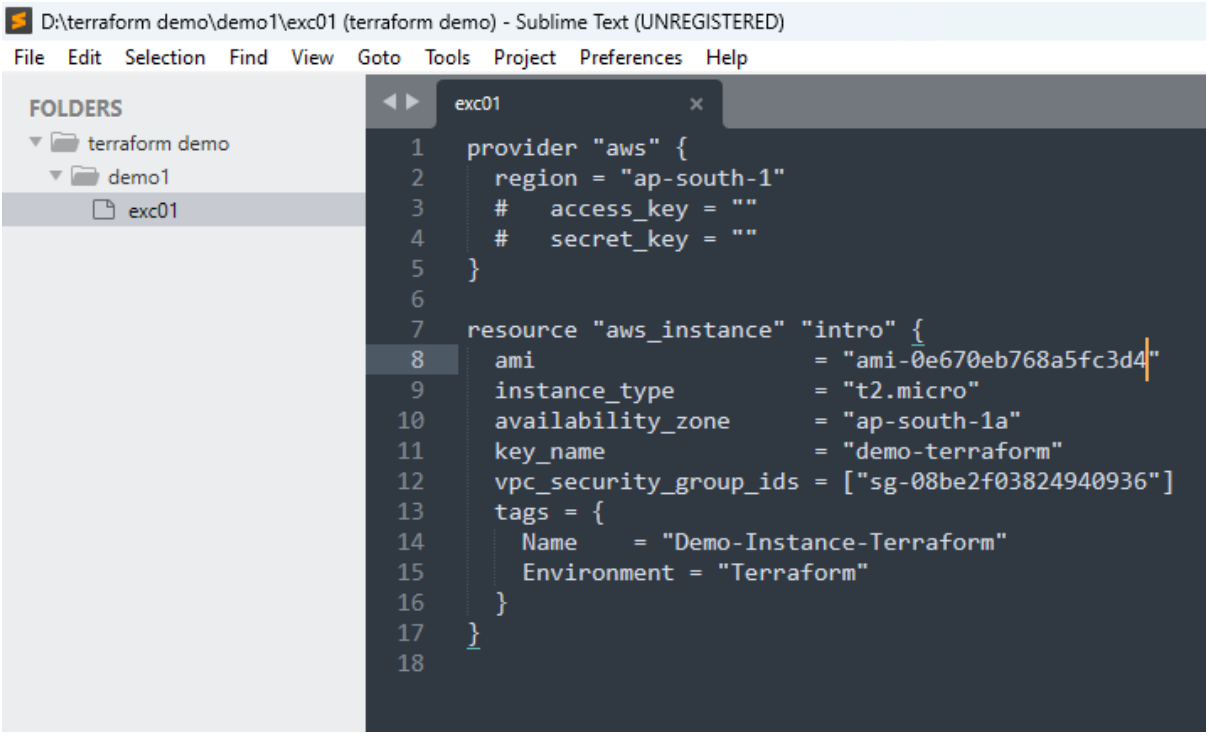
12. Then you are going to right click on the folder that you've opened and click on new folder again. You are going to create a folder into a folder. Now give it a name.



13. Once you are done then again right click on the new folder and then click on new file. This will create a new file for you to write your code.
14. In that you are going to write your code and execute it.



15. Now there are some pre-requisites for you. You need to **create a security group then copy the security group ID** and a **key pair.** You also need to copy the **AMI ID** of what OS you want for terraform to launch your instance with.
16. You will also need an **IAM user and configure** it on your local machine.
17. Now below is the code which you can see in the snapshot. In this code you are required to change the AMI ID according to your region, the key name as per your key. Change the security group ID too.
18. And then save it with **extension .tf**



19. Once you are done then open gitbash and navigate to the drive and folder where you have created your file.
20. Then if you do a listing for your files you can see your file there.

```
LAPTOP-G2CAKBK8 MINGW64 /d/terraform-demo/demo1
$ ls
exc01.tf
```

21. If you want to view your file you can also do that.

```
$ cat exc01.tf
provider "aws" {
  region = "ap-south-1"
  #   access_key = ""
  #   secret_key = ""
}

resource "aws_instance" "intro" {
  ami                    = "ami-0e670eb768a5fc3d4"
  instance_type          = "t2.micro"
  availability_zone      = "ap-south-1a"
  key_name               = "demo-terraform"
  vpc_security_group_ids = ["sg-08be2f03824940936"]
  tags = {
    Name    = "Demo-Instance-Terraform"
    Environment = "Terraform"
  }
}
```

22. So, I'm just going to run **terraform init** and this is going to check my provider that is AWS and it's going to download the plugins for AWS in the current working directory.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.38.0...
- Installed hashicorp/aws v5.38.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

23. Before we really execute, we are going to do some validation for code. We'll say **terraform validate** which will syntactically check our code. It will give you some errors if there are.

```
        LAPTOP-G2CAKBK8 MINGW64 /d/terraform-demo/demo1
$ terraform validate
Success! The configuration is valid.
```

24. There's also one more option you can use after this FMT format. So, it will format your terraform code, or should I say rewrite what it says canonical format and style.

```
LAPTOP-G2CAKBK8 MINGW64 /d/terraform-demo/demo1
$ terraform fmt
exc01.tf

LAPTOP-G2CAKBK8 MINGW64 /d/terraform-demo/demo1
$ cat exc01.tf
provider "aws" {
  region = "ap-south-1"
  #   access_key = ""
  #   secret_key = ""
}

resource "aws_instance" "intro" {
  ami                    = "ami-0e670eb768a5fc3d4"
  instance_type          = "t2.micro"
  availability_zone      = "ap-south-1a"
  key_name               = "demo-terraform"
  vpc_security_group_ids = ["sg-08be2f03824940936"]
  tags = {
    Name        = "Demo-Instance-Terraform"
    Environment = "Terraform"
  }
}
```

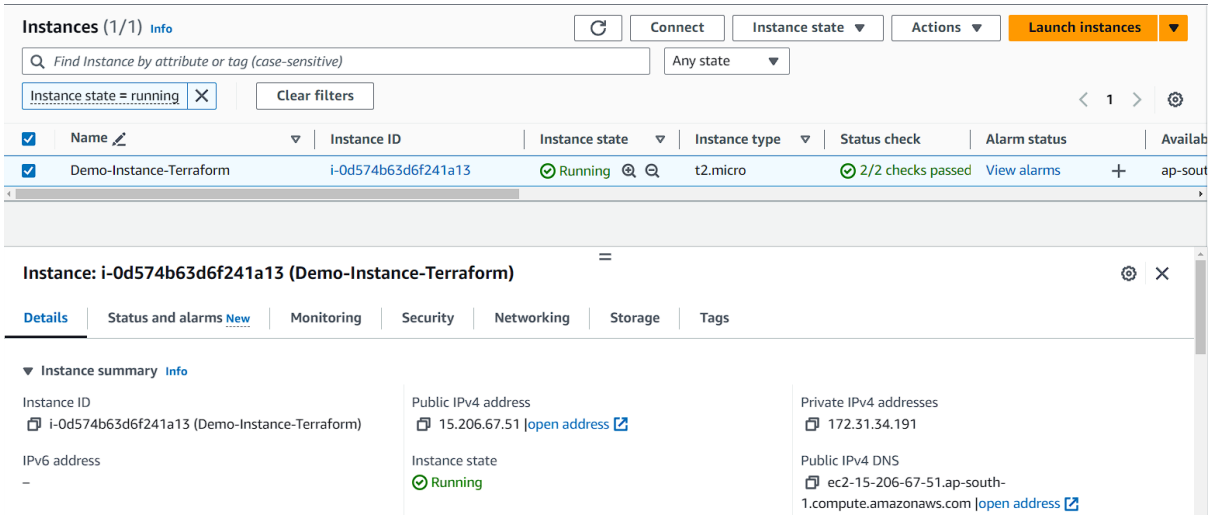
25. Now **terraform plan** is not going to do anything. It's just going to show you that what will be the execution. It's just going to give you like a preview what will happen if you apply this.

```
      + tenancy                                = (known after apply)
      + user_data                              = (known after apply)
      + user_data_base64                       = (known after apply)
      + user_data_replace_on_change            = false
      + vpc_security_group_ids                 = [
          + "sg-08be2f03824940936",
        ]
    }

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

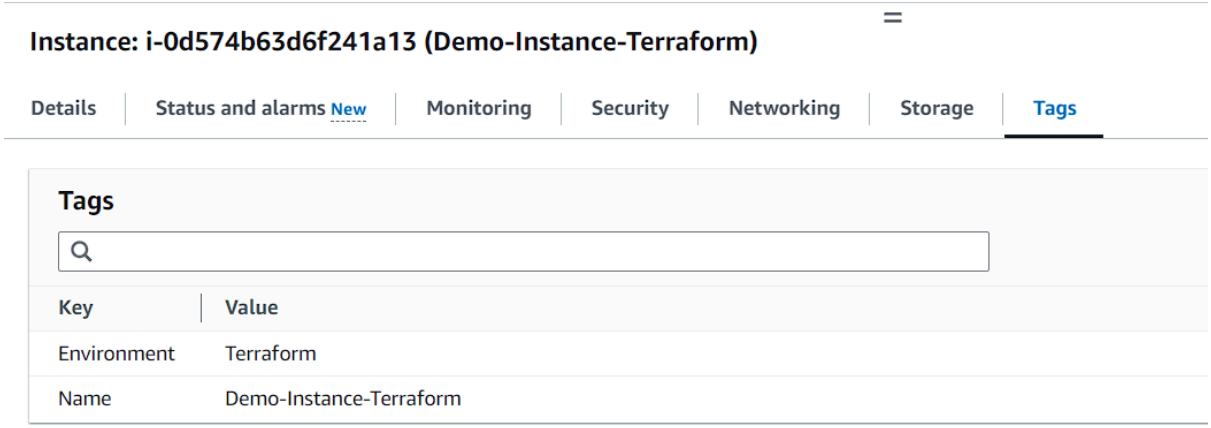
26. The final command is **terraform apply.** Now this command will apply everything and it will create your instance.
27. Once you've executed this command it will ask your whether you want to this to happen or not. So, just type yes and click on enter.
28. And here you can see that it has started creating the instance and the creating is complete.

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.intro: Creating...
aws_instance.intro: Still creating... [10s elapsed]
aws_instance.intro: Still creating... [20s elapsed]
aws_instance.intro: Still creating... [30s elapsed]
aws_instance.intro: Creation complete after 32s [id=i-0d574b63d6f241a13]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

29. Now if you will navigate to AWS Console and then to EC2. You can see your instance up and running.



30. You can also check for the tags.



31. Now once you are done just write terraform destroy in your gitbash and it will destroy everything.

```
Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_instance.intro: Destroying... [id=i-0d574b63d6f241a13]
aws_instance.intro: Still destroying... [id=i-0d574b63d6f241a13, 10s elapsed]
aws_instance.intro: Still destroying... [id=i-0d574b63d6f241a13, 20s elapsed]
aws_instance.intro: Still destroying... [id=i-0d574b63d6f241a13, 30s elapsed]
aws_instance.intro: Still destroying... [id=i-0d574b63d6f241a13, 40s elapsed]
aws_instance.intro: Destruction complete after 41s

Destroy complete! Resources: 1 destroyed.
```