Terraform: A Practical Starter Guide

This guide was inspired by the helpful tutorials and resources from the cloud community at abcofcloudcomputing.com.

What is Terraform and Infrastructure as Code?

Imagine you are building a custom house. Traditionally, you would have to manually lay every brick, install every pipe, and paint every wall by hand. If you wanted to build an exact copy of that house somewhere else, you would have to repeat the entire manual process from scratch.

In the software world, **Infrastructure as Code (IaC)** acts like a master architect's digital blueprint. Instead of manually clicking buttons in a cloud provider's web console to create servers or databases, you write down instructions in simple code files. **Terraform** is the "Master Builder." It reads your blueprint file, understands what needs to be created or updated, and automatically builds everything for you accurately every single time.

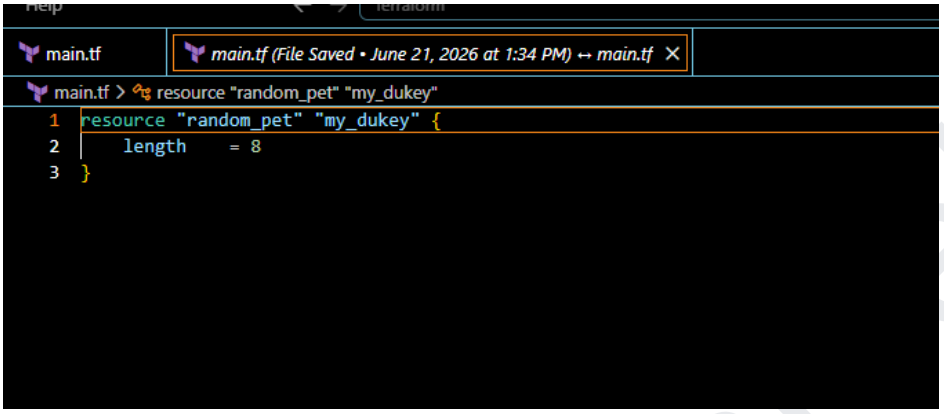
Key Concepts & Terms

Term	Beginner Explanation (Analogy)
Resource	An individual component in your blueprint (e.g., a specific room, a server, or a pet name generator).
State File (terraform.tfstate)	The Master Builder's ledger/notebook. It tracks what has actually been built so far in the real world.
Variables	Customizable placeholders in your blueprint (e.g., choosing a default wall color like "Blue").
Variables File (terraform.tfvars)	A special order form used to override defaults with specific custom values (e.g., changing wall color to "Red").

Sarah Daniel

Task 1:

Create a Random_pet resourcegive it value, run terraform apply then show configuration.



Understanding the Configuration Syntax

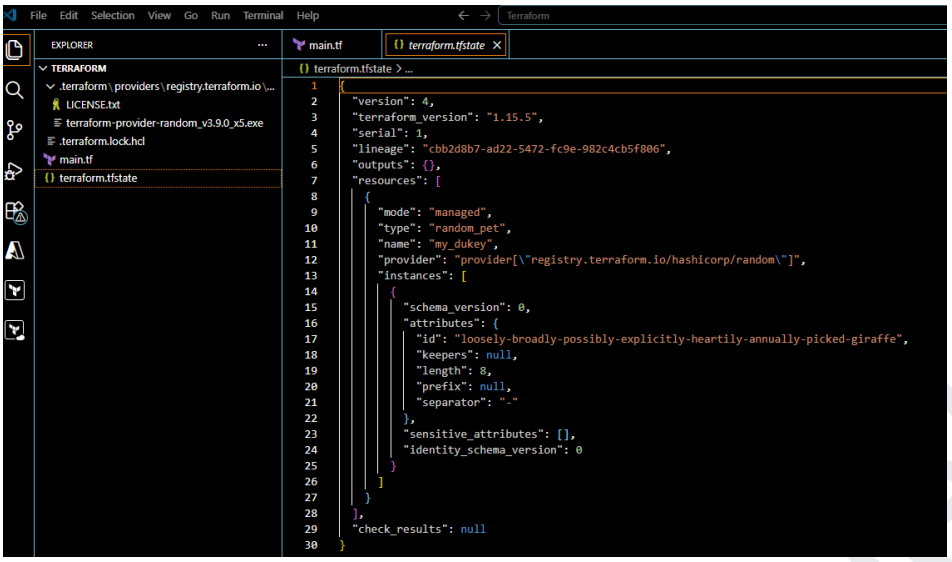
In the screenshot above, we defined our very first Terraform resource in a file called **main.tf**. We use the **random_pet** resource provider—a fun, harmless provider that generates random, human-readable pet names (like "swift-dog" or "happy-cat"). It's the perfect tool for practice because it doesn't cost money or create complex infrastructure!

Let's break down the code syntax line-by-line:

- **resource "random_pet" "my_dukey"**: **random_pet** is the *resource type* (what kind of block we are building), while **"my_dukey"** is the *local name* we give this specific block inside our blueprint.
- **length = 8**: This is an *argument* or instruction setting the rule that our generated pet name must consist of 8 random words linked together.

Show terraform state file.

Sajan Paul



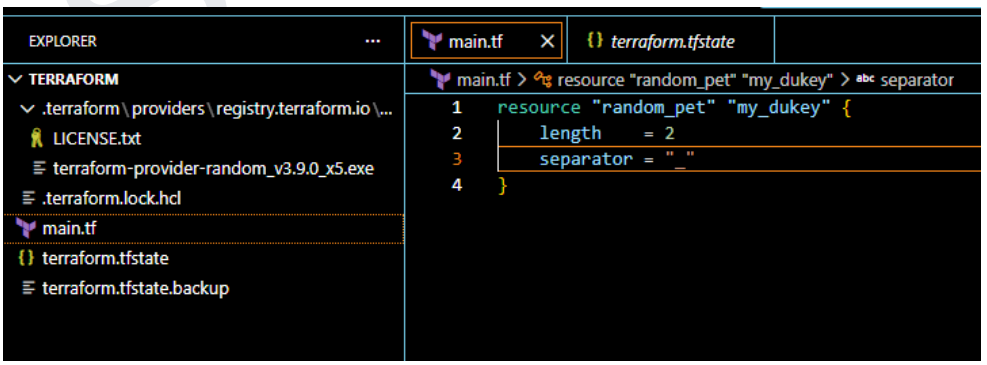
Reading the State File Ledger

After running **terraform apply**, Terraform creates a file named **terraform.tfstate**. Think of this JSON file as the Master Builder's official inventory ledger.

Looking closely at the screenshot above, you can read the recorded attributes of the created resource:

- **"type": "random_pet"** and **"name": "my_dukey"**: Identifies which resource in our code this record belongs to.
- **"id": "loosely-broadly-possibly-explicitly-heartily-annually-picked-giraffe"**: This long, 8-word hyphenated string is the unique identifier generated by Terraform according to our length rule!
- **"length": 8**: Confirms the rule configured during the creation.

Change value, run apply, then show the current state file.

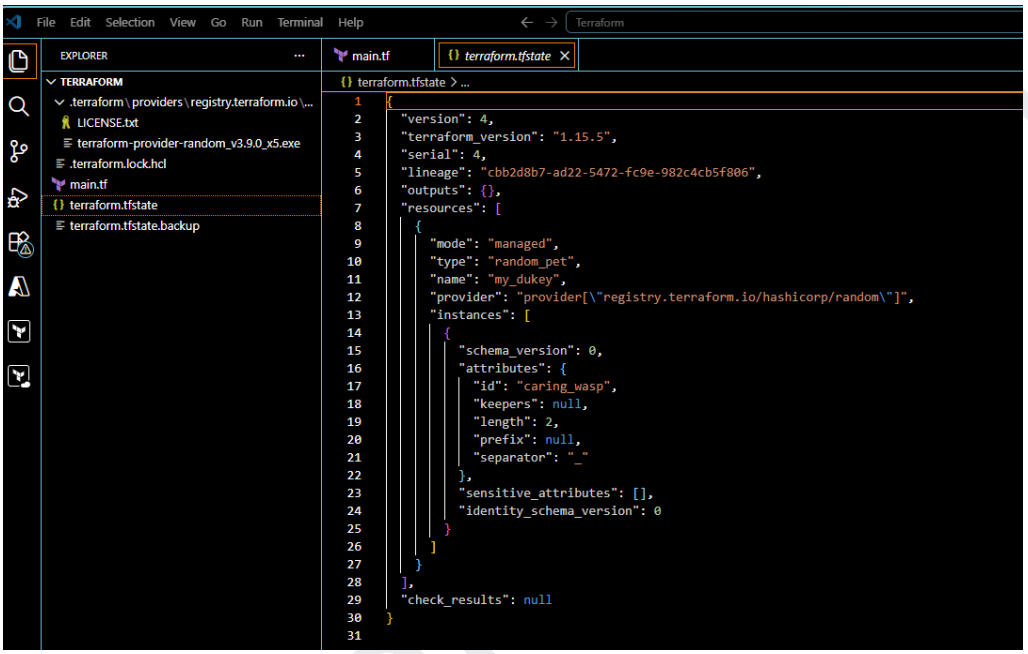


Samh Daniel

Modifying the Configuration

In this step, we updated our blueprint in **main.tf** by changing **length = 2** and adding **separator = "_"**. When we run **terraform apply** again, Terraform compares our new blueprint with the state file ledger. It notices the requested changes and replaces the old 8-word pet name with a new 2-word pet name separated by an underscore instead of a hyphen.

Updated state file.



Verifying the Updated State

Opening the updated **terraform.tfstate** confirms that our changes took effect successfully:

- **"id": "caring_wasp"**: The new pet name generated consists of exactly 2 words separated by an underscore.
- **"length": 2** and **"separator": "_"**: Reflects the updated arguments from our configuration.

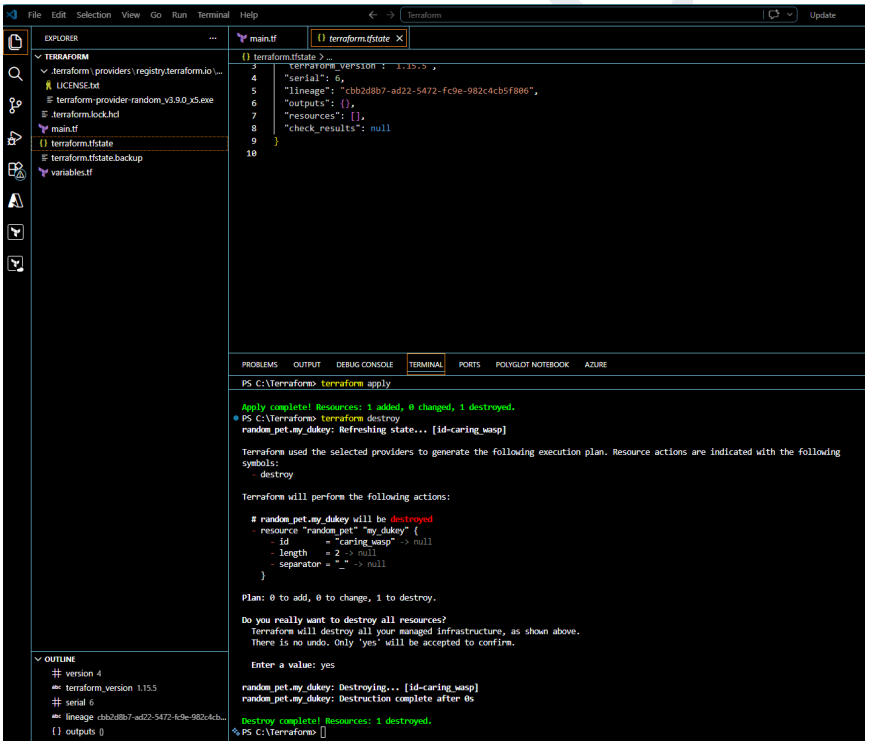
Task 2:

In task 2, we introduce **Variables** to make our configuration reusable and flexible. Instead of hardcoding values directly into **main.tf**, variables act as flexible input slots. Think of variables as options on a house building template—allowing you to easily swap paint colors without rewriting the structural design. We can specify default values in **variables.tf** or override them using a **terraform.tfvars** file.

Using the values created in assignment1, create a variable, name the default variables.
Proceed to apply with the default variable.Then create a teraaform.tfvars, change the values.
Show the terraform plans and show the state file for the default variable and .tfvars.

Daniel

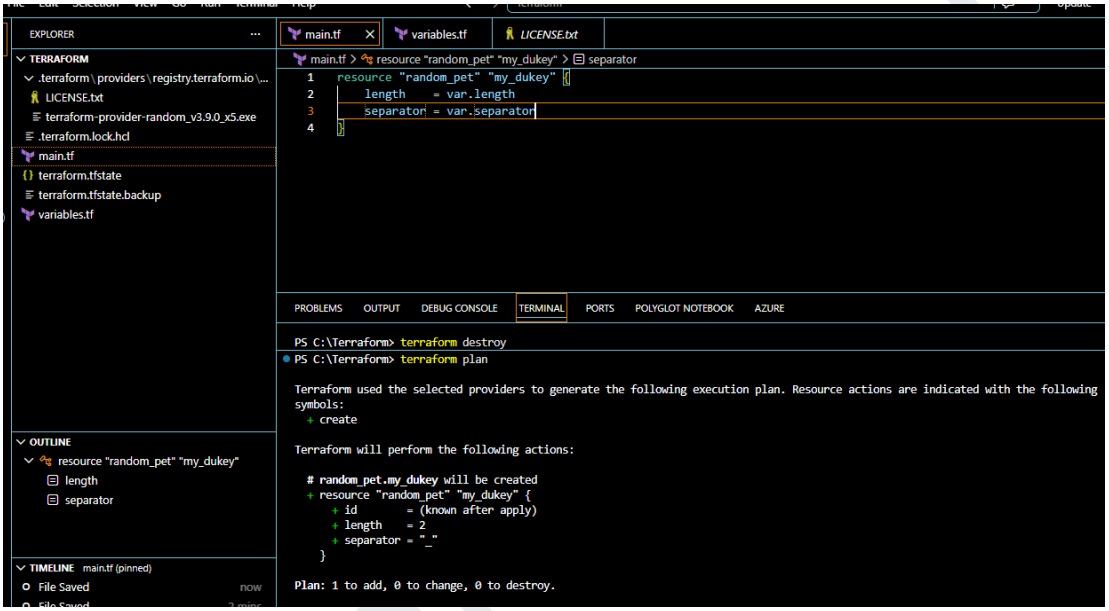
Terraform Destroy



Understanding Terraform Destroy

Before starting Assignment 2 fresh, we execute **terraform destroy**. This command initiates the teardown phase. Terraform checks the state file to see what exists, displays a plan indicating which resources will be destroyed (shown with red minus signs in terminal output), and prompts you for safety confirmation typing **yes** before wiping resources clean.

Showing the default variable plan.

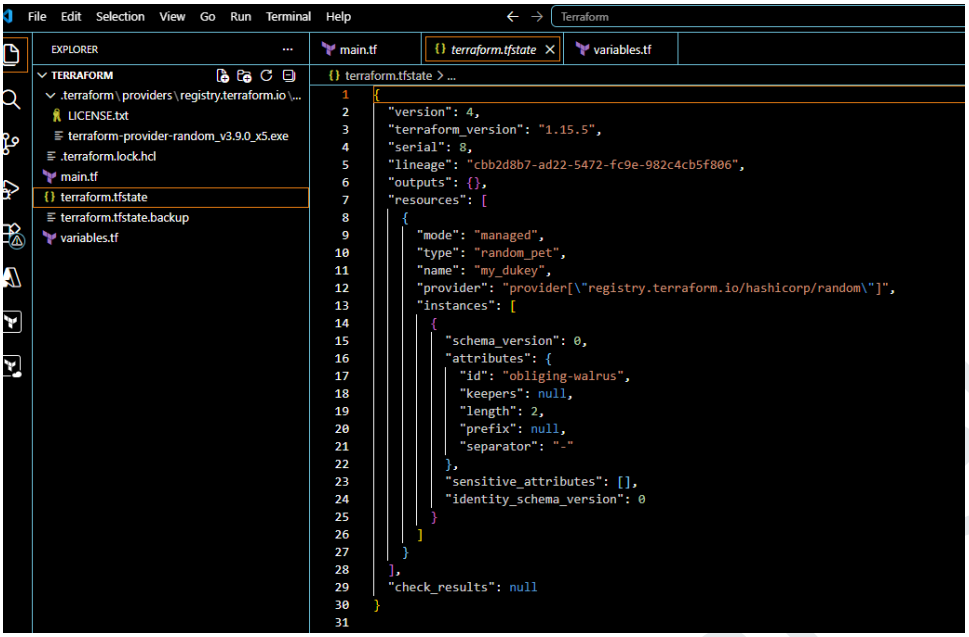


Reading a Terraform Execution Plan

When running **terraform plan**, Terraform performs a dry run to preview what changes will happen without making actual modifications yet. In the screenshot:

- **Green plus (+)**: Indicates a new resource will be created.
- **var.length & var.separator**: Shows that **main.tf** now references variables instead of static hardcoded numbers.
- **Plan: 1 to add, 0 to change, 0 to destroy**: Gives a clear summary of upcoming actions.

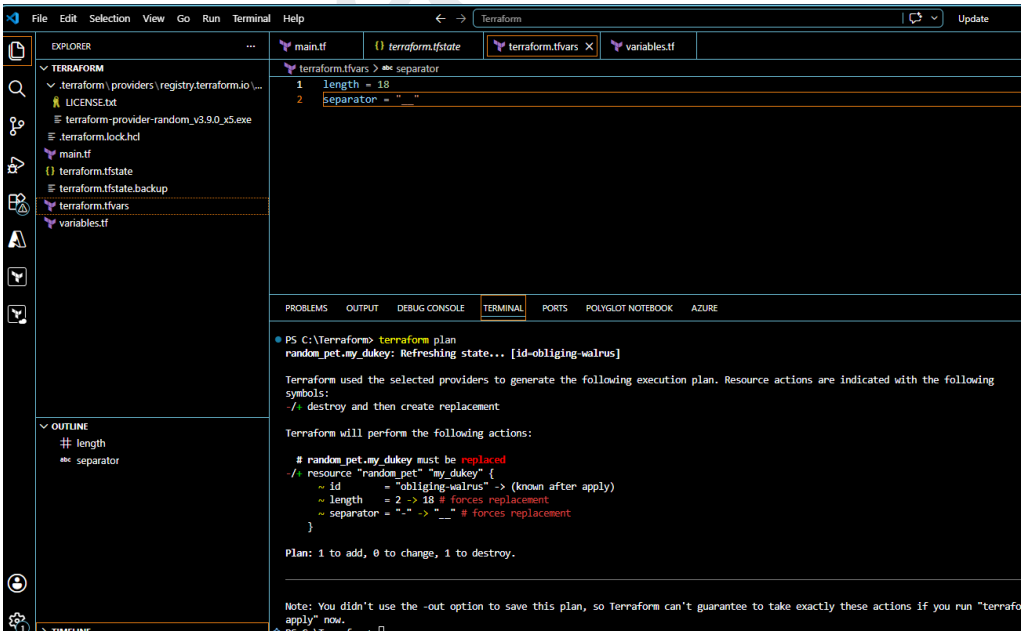
The default variables state file.



Default Variable Application

Applying the configuration created a resource using the default variable values specified in **variables.tf**. As recorded in the state file ledger above, it generated an ID **"obliging-walrus"** with length 2 and hyphen separator.

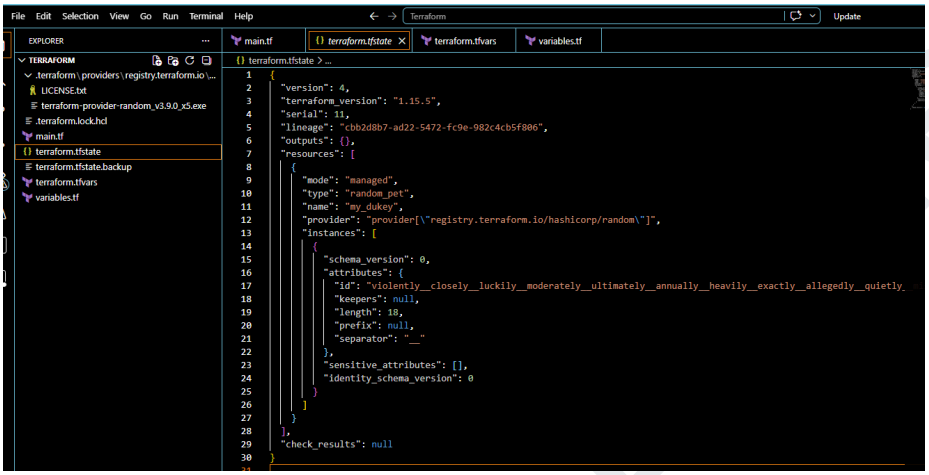
Showing the *.tfvars* file plan



Understanding "Forces Replacement"

In this step, we created a **terraform.tfvars** file setting **length = 18** and **separator = "__"** (double underscore). Notice the execution plan symbol **-/+ destroy and then create replacement**. Because random pet names cannot simply be edited in-place once created, Terraform must completely destroy the old 2-word pet name and construct a brand-new 18-word pet name resource from scratch.

Showing the *.tfvars* state file



Final State Verification

Looking at the final **terraform.tfstate** ledger, we can see the full 18-word pet name separated by double underscores (e.g., **"violently__closely__luckily..."**), confirming that our custom values from **terraform.tfvars** successfully took precedence!