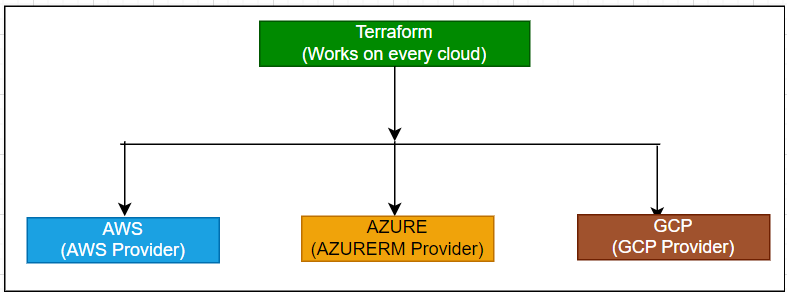
**Terraform**

**Introduction:**  Terraform is an opensource Infra as a code (IAC) tools which is created by Hashicorp. By using this tool we can automate our cloud infra like RG, storage account , VM etc creation.

****

**Provider:** Provider is just like a plugins that allows terraform to manage resources on different platforms like aws, azure, gcp etc. Providers let Terraform know which cloud service to communicate with for creating or managing resources.

**AzureRM Provider (azurerm) :** The Azure Resource Manager (AzureRM) provider is a Terraform plugin that allows users to manage Azure resources. It interacts with Azure Resource Manager APIs to create, manage, and delete resources like virtual machines, storage accounts, and network interfaces.

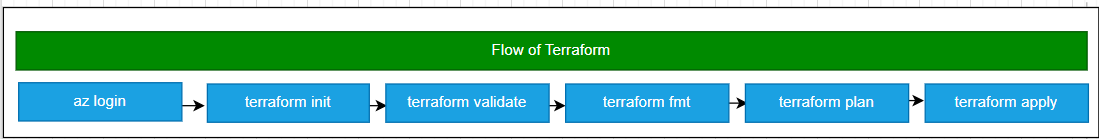
**Key Concepts of AzureRM Provider:**

* **Configuration:** Setting up the provider requires specifying necessary credentials and settings to authenticate with Azure.
* **Resources:** Azure resources such as Virtual Machines, Resource Groups, Storage Accounts, etc., that can be managed through Terraform.
* **Data Sources:** Allows querying existing resources and using the data for other configurations.

Note: Need to run terraform commands where provider.tf file present

****

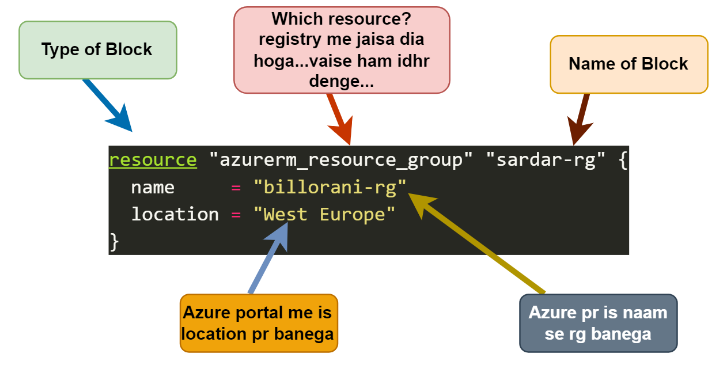
**Flow of Terraform:**

****

**A yellow square with black text

Description automatically generated**

**Terraform resource Block Structure:**

****

**Terraform State File:** The Terraform state file is a special file where Terraform keeps track of the resources it manages. It stores information about your infrastructure, such as which resources exist, their current settings, and any changes made.

* Terraform state file is a JSON-formatted file that keeps track of the resources managed by Terraform.
* It stores metadata about the resources, their dependencies, and their properties.
* The state file is crucial for Terraform to understand the current state of the infrastructure and to plan and apply changes accurately.

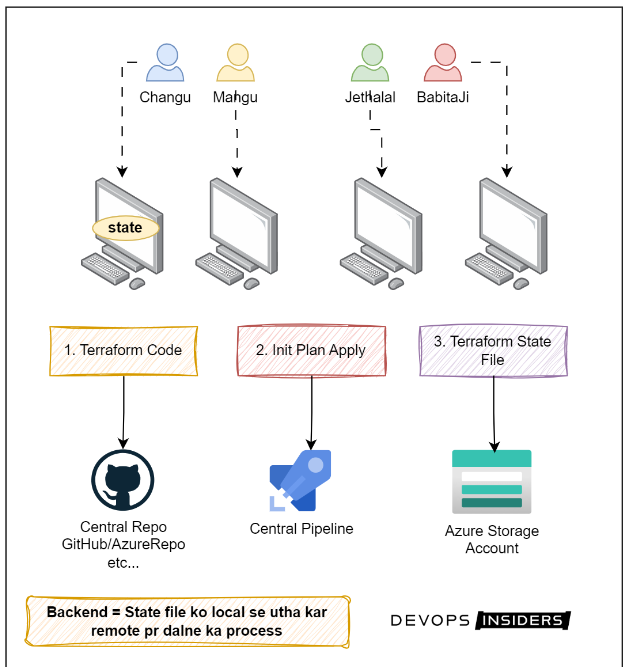
Key Points:

* Tracking: It helps Terraform understand what resources it has created and their current state.
* Updates: When you make changes and run Terraform again, it compares the state file with your configuration to know what needs to be updated.
* Location: By default, this file is saved locally on your machine, but it can also be stored remotely (like in Azure Storage) for better collaboration.

In short, the state file is crucial for Terraform to manage and keep your infrastructure in sync with your code. We cant edit terraform state file.

**Terraform State Commands**

1. terraform state list # Lists all resources recorded in the state file.
2. terraform state show # Displays detailed information about a specific resource in the state file
3. terraform state mv #  Moves a resource from one part of the state file to another.
4. terraform state rm # Removes a resource from the state file, without affecting the real-world resource.
5. terraform state pull # Pulls the state from the remote backend and outputs it to stdout.
6. terraform state push # Pushes the local state to the remote backend.

****

**Setting Remote Backend in Azure Storage Account :**

**A screen shot of a computer program

Description automatically generated**

**Note:** Need to create storage account and container in backend before proceeding terraform init command

* This configuration sets up Azure Storage Account as the remote backend for Terraform.
* It specifies the Azure resource group, storage account, container, and key (filename) for storing the state file remotely.

**Terraform State Locking**

* Terraform state locking is a mechanism to prevent concurrent operations on a Terraform state file.
* This helps to avoid conflicts and ensure consistency when multiple users or processes are working with the same state file.

How does it work?

* When Terraform performs a state-changing operation (like terraform apply or terraform destroy), it attempts to acquire a lock.
* If the lock is acquired successfully, the operation proceeds. If not, Terraform waits or fails depending on the configuration.
* After the operation completes, the lock is released.

Types of backends that support locking:

* S3 with DynamoDB
* Azure Blob Storage
* Google Cloud Storage
* HashiCorp Consul

**State Locking with Azure Storage Account**

* Azure Storage Account can be used as a backend for storing Terraform state files.
* Azure Blob Storage supports state locking and consistency checking via Azure Blob Lease mechanism.

**A screenshot of a computer

Description automatically generated**

When two people run terraform command simultaneously, then how state file works?

When two people run Terraform commands simultaneously, the state file locks to prevent conflicts. If locking is enabled, the second person must wait until the first finishes; otherwise, they might overwrite each other's changes.

**Acquire lease:** When we run a Terraform command (like applying changes), it tries to "acquire a lease" on the state file stored in Azure Blob Storage.

Why It Matters: This lease acts like a lock. It prevents anyone else from making changes to the infrastructure at the same time, which helps avoid mistakes or conflicts.

How It Works: If Terraform gets the lease, it can safely make updates. The lease lasts for a short time (usually around 15-60 seconds).

**Break Lease:** "Break lease" means forcibly releasing the lock on the state file.

Why You Might Need It: If a process that has the lease crashes or is stuck, you might need to break the lease so that you or someone else can run Terraform commands again.

How to Do It: You can use the command terraform force-unlock <LOCK\_ID> to break the lease. Just be careful, as this can lead to problems if another operation is still in progress.

**Tips for Using Terraform with Azure:**

* Always use Azure Blob Storage for your Terraform state, as it supports locking.
* Make sure no other Terraform commands are running before breaking a lease.
* Only force unlock if you are sure it’s necessary.
* This way, you can safely manage your infrastructure without running into issues!

**Acquire Lease (लीज़ हासिल करना)**

यह क्या है: जब आप Terraform कमांड (जैसे कि बदलाव लागू करना) चलाते हैं, तो यह Azure Blob Storage में स्टेट फ़ाइल पर "लीज़ हासिल करने" की कोशिश करता है।

यह क्यों महत्वपूर्ण है: यह लीज़ एक ताले की तरह काम करती है। यह सुनिश्चित करती है कि कोई और एक ही समय में इन्फ्रास्ट्रक्चर में बदलाव न कर सके, जिससे गलतियों या संघर्षों से बचा जा सके।

यह कैसे काम करता है: यदि Terraform लीज़ हासिल कर लेता है, तो वह सुरक्षित रूप से अपडेट कर सकता है। लीज़ एक छोटे समय के लिए (आमतौर पर 15-60 सेकंड) रहती है।

**Break Lease (लीज़ तोड़ना)**

यह क्या है: "लीज़ तोड़ना" का मतलब है स्टेट फ़ाइल पर ताला को मजबूरन खोलना।

आपको इसकी आवश्यकता क्यों हो सकती है: यदि कोई प्रक्रिया जो लीज़ रखती है क्रैश हो जाती है या फंस जाती है, तो आपको लीज़ तोड़ने की जरूरत हो सकती है ताकि आप या कोई और फिर से Terraform कमांड चला सके।

इसे कैसे करें: आप terraform force-unlock <LOCK\_ID> कमांड का उपयोग करके लीज़ तोड़ सकते हैं। बस ध्यान रखें, क्योंकि यह समस्याएं पैदा कर सकता है यदि कोई और प्रक्रिया अभी भी चल रही है।

**Azure के साथ Terraform का उपयोग करते समय सुझाव**

हमेशा Terraform स्टेट के लिए Azure Blob Storage का उपयोग करें, क्योंकि यह लॉकिंग का समर्थन करता है।

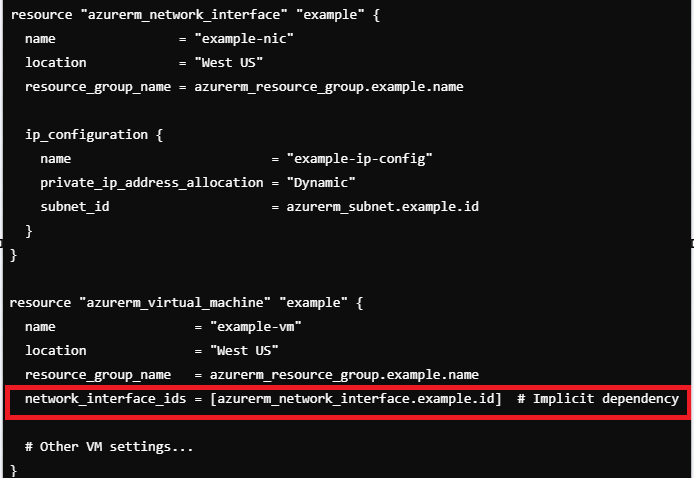
लीज़ तोड़ने से पहले सुनिश्चित करें कि कोई और Terraform कमांड चल नहीं रही है।

केवल तभी लीज़ को मजबूरन खोलें जब आप सुनिश्चित हों कि इसकी आवश्यकता है।

इस तरह, आप अपनी इन्फ्रास्ट्रक्चर को सुरक्षित रूप से प्रबंधित कर सकते हैं बिना किसी समस्या के!

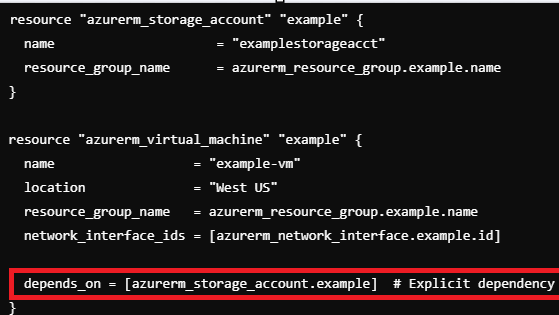
**Implicit Dependency:** This happens automatically when one resource uses another resource's output. Automatically determined by Terraform.

Example: A virtual machine depends on a network interface.

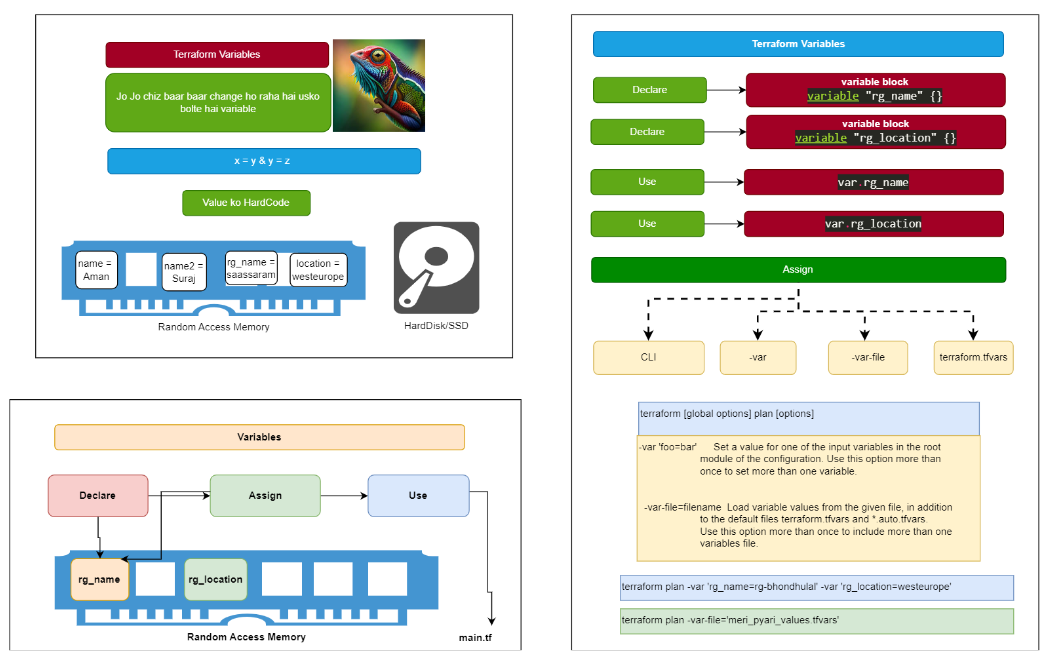


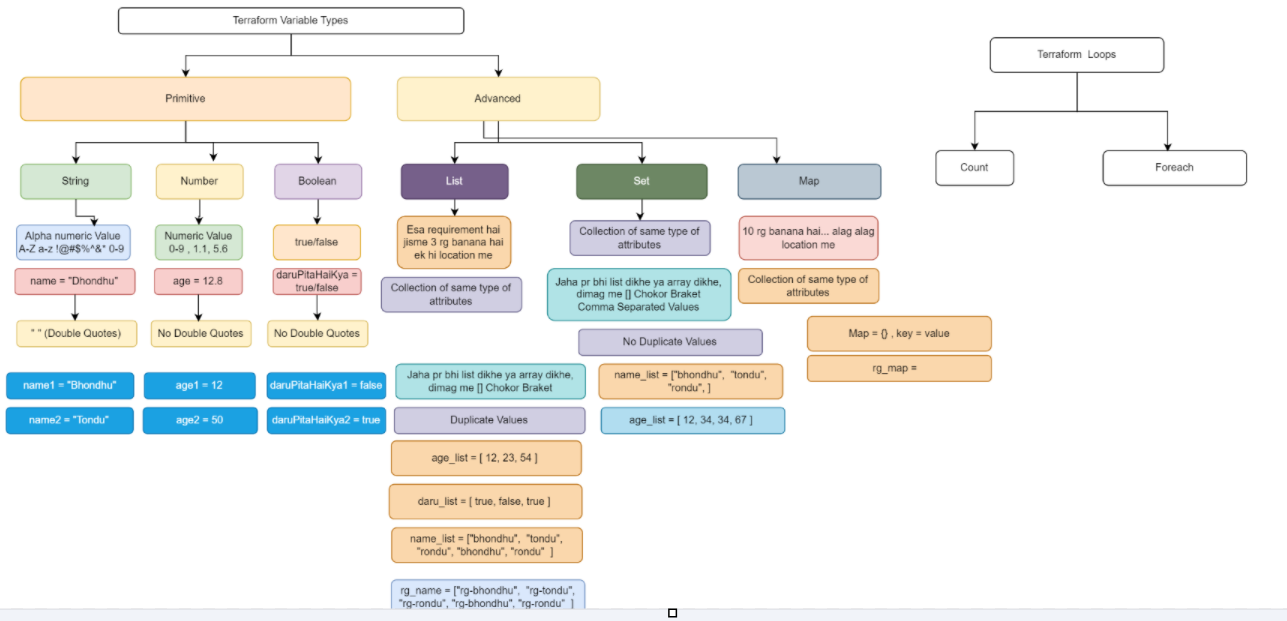
**Explicit Dependency:** This is when we manually tell Terraform that one resource depends on another using the depends\_on argument. Manually defined using depends\_on.

Example: we manually specify that the virtual machine should wait for the storage account to be created first.

****

**Terraform Variable:** In Terraform, a variable is a placeholder for a value that you can set and reuse throughout your configuration. It allows you to make your Terraform code flexible and customizable.

****



**List :**

* A list in Terraform is an ordered collection of values identified by their index.
* Lists are useful when you need to maintain a specific order of items.
* Declaring a list variable in Terraform:

A computer code with black text

Description automatically generated with medium confidence

* Here, rg\_names is a list of strings containing four resource group names.

**Set:**

* A set in Terraform is an unordered collection of unique values.
* Sets ensure that each item in the collection is unique and can be useful when the order of items is not important.
* Converting a list to a set can be done using the toset function.

**Resources**

**For\_each with toset**

* The for\_each argument allows you to iterate over collections and create multiple resources based on the collection.
* The toset function is used here to convert the list var.rg\_names into a set, ensuring unique and unordered values for iteration.



* In this resource block, the for\_each statement iterates over the set created from the rg\_names list.
* each.value represents the current value from the set being iterated over.

**Explanation:**

1. **Declaring the Variable:**
   * The variable rg\_names is declared as a list of strings with default values ["rg1", "rg2", "rg3", "rg4"].
2. **Converting List to Set:**
   * The toset function is used to convert the list var.rg\_names into a set, ensuring that each resource group name is unique and unordered.
3. **Iterating with for\_each:**
   * The for\_each argument is used in the azurerm\_resource\_group resource to iterate over the set of resource group names.
   * For each unique value in the set, a resource group is created with the name specified by each.value and located in "West Europe".

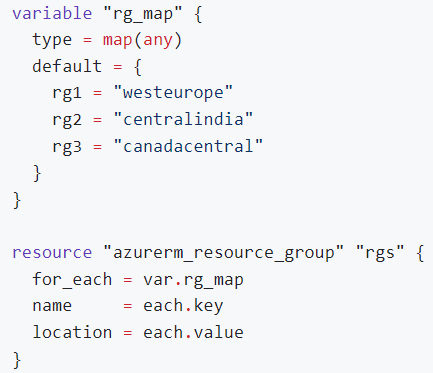
**Key Points:**

List: Ordered collection, allowing duplicate values.

Set: Unordered collection of unique values, created here using toset.

for\_each: Iterates over collections, creating multiple resources based on the items in the collection.

**Terraform Map :** A map in Terraform is a collection of key-value pairs, where each key is unique. Maps are useful for defining related sets of data that you can iterate over using constructs like for\_each. In this example, we will look at how to use maps and the for\_each construct to create multiple Azure resource groups and storage accounts.

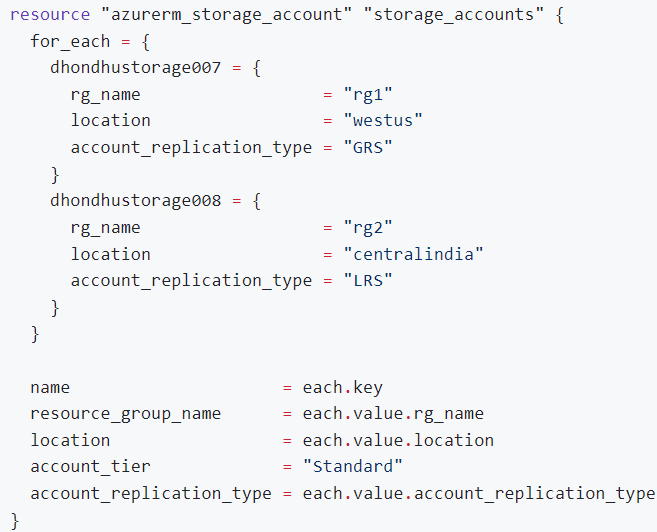
****

Explanation:

1. Variable Declaration:
   * A variable rg\_map of type map(any) is declared. The default value of this map contains three key-value pairs, each representing a resource group name and its corresponding location.
2. Resource Group Creation:
   * The azurerm\_resource\_group resource uses for\_each to iterate over the map var.rg\_map.
   * For each entry in the map:
     + name is set to the key of the map entry (e.g., rg1).
     + location is set to the value of the map entry (e.g., westeurope).

This will create three resource groups with the names and locations defined in the rg\_map.

**Terraform for\_each with Nested Maps:** Terraform's for\_each construct can also be used to iterate over nested maps. This is useful when creating resources with multiple properties.



1. **Resource Definition:**
   * The azurerm\_storage\_account resource uses for\_each to iterate over a nested map, where each key is the name of the storage account and the value is a map containing the properties of the storage account.
2. **Resource Properties:**
   * name is set to the key of the map entry (e.g., dhondhustorage007).
   * resource\_group\_name is set to the rg\_name property of the map value.
   * location is set to the location property of the map value.
   * account\_tier is set to a constant value Standard.
   * account\_replication\_type is set to the account\_replication\_type property of the map value.

This will create two storage accounts with the names and properties defined in the map.

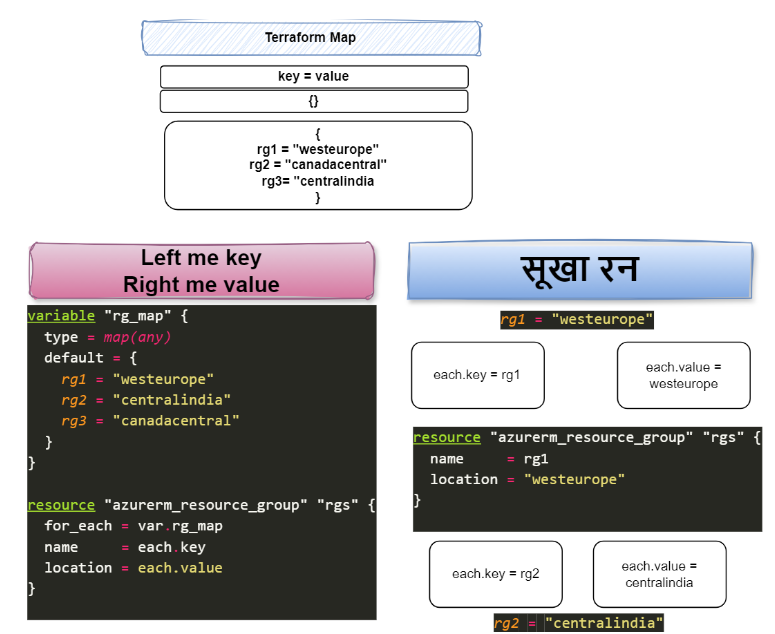
**Summary**

Maps in Terraform are useful for managing collections of related data.

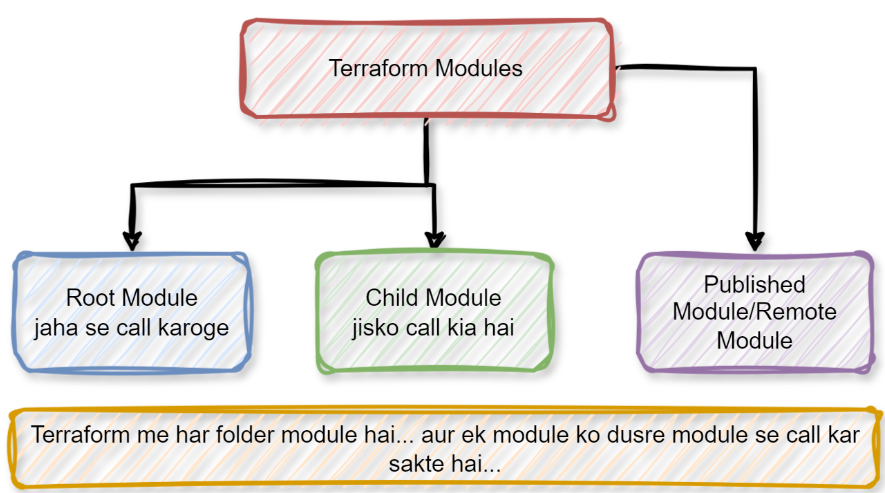
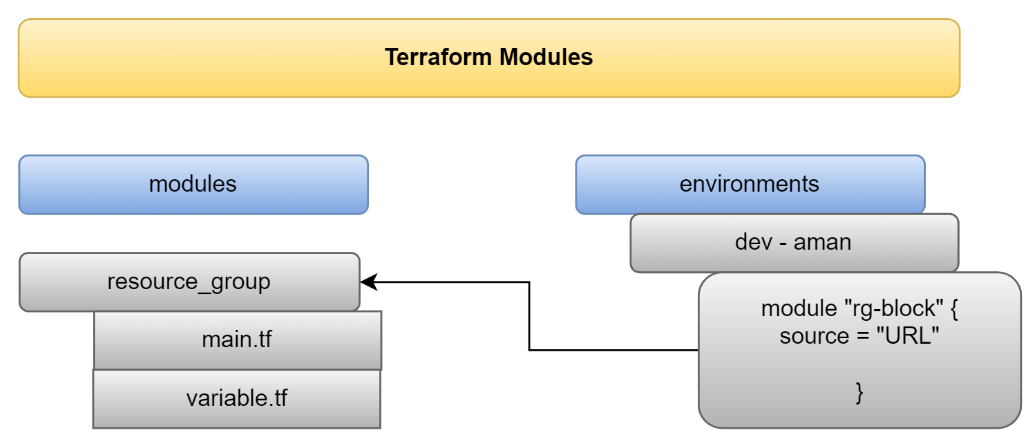
The for\_each construct allows you to iterate over maps to create multiple resources.

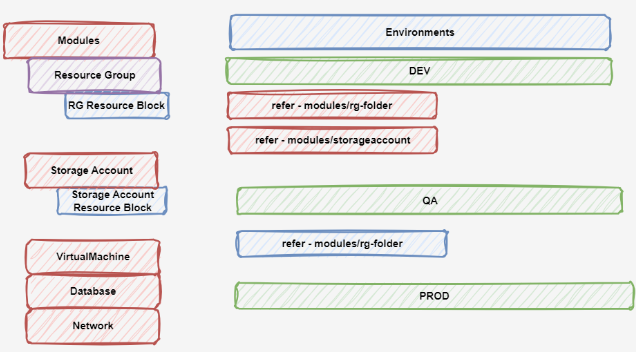
Using maps with for\_each helps in managing and scaling your Terraform configurations efficiently, reducing repetition and improving readability.

These examples demonstrate how to use maps and for\_each to dynamically create multiple resources with different properties in Terraform.



**Terraform Module:**



A screenshot of a computer program

Description automatically generated

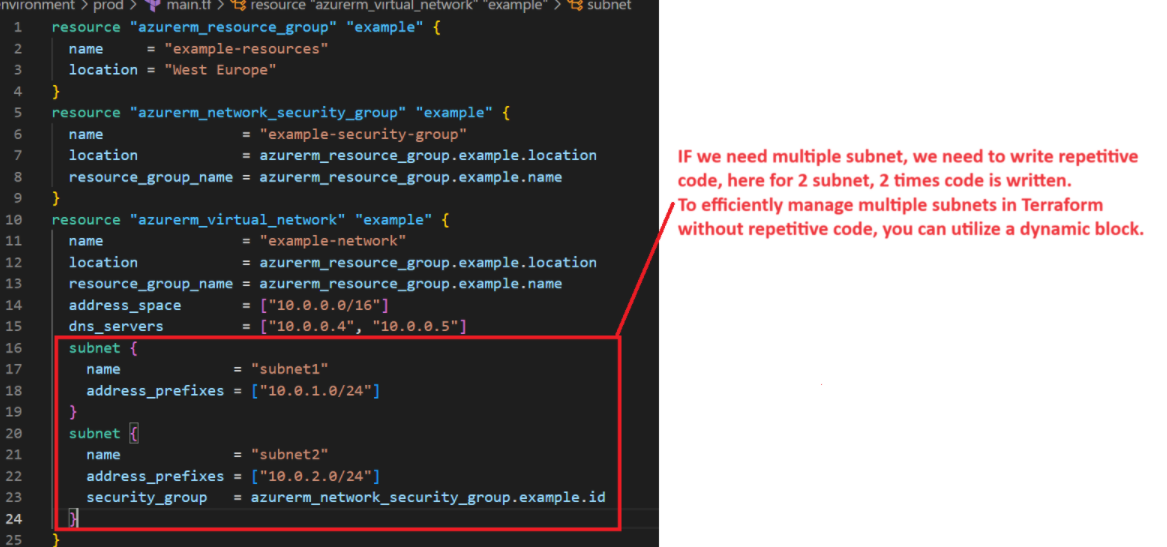
**Terraform Dynamic Block:**

Dynamic blocks help us to avoid writing the same code multiple times. Instead of duplicating similar configurations, we can use a dynamic block to create them in a loop.

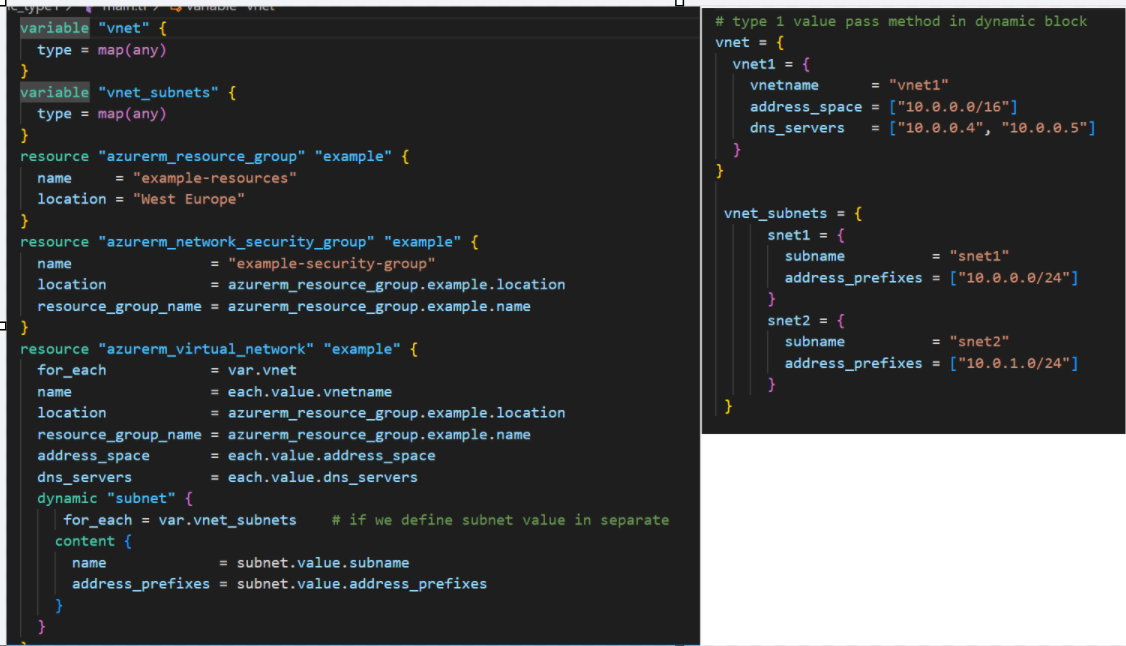
For instance, if you want to create multiple firewall rules for different ports, a dynamic block can iterate through a list of ports and create a rule for each one, all within the same resource block.

**Key Points:**

* **Reduces Repetition**: Avoids writing the same configuration multiple times.
* **Increases Flexibility**: Easily adapts to changes by modifying the input list.
* **Cleaner Code**: Makes your configuration easier to read and maintain.



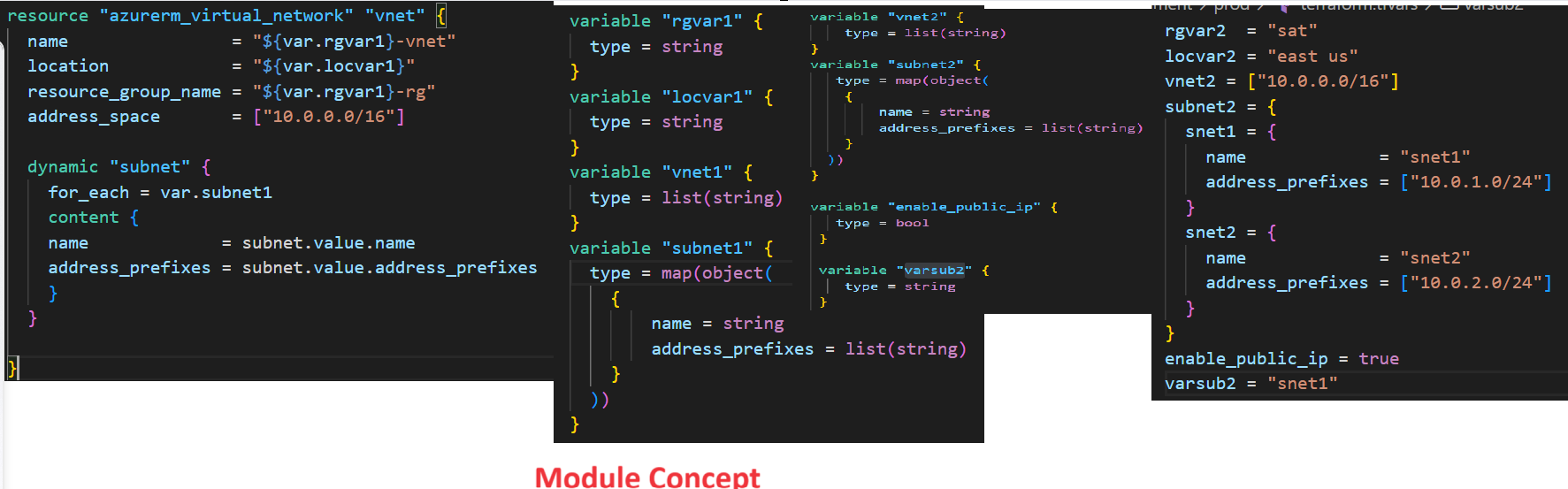
Type 1:



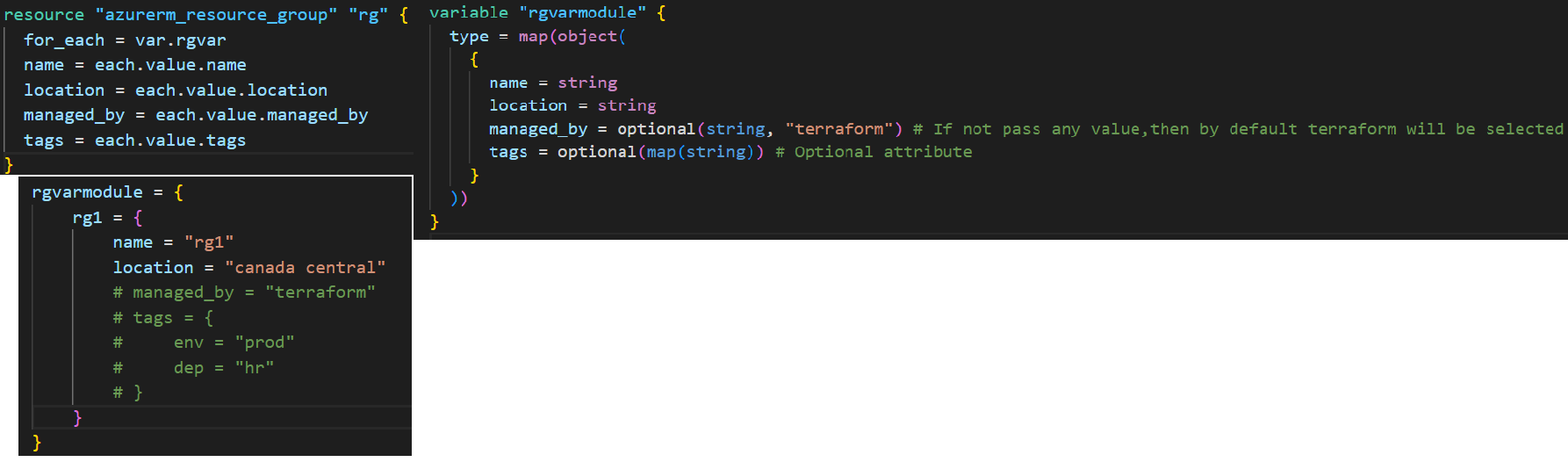
Type 2:



Type 3:



**Terraform optional Attributes:**



**Terraform Interpolation:**

Interpolation in Terraform for Azure is a way to insert values from variables or resources into our configuration. It allows you to create dynamic and flexible setups by combining different pieces of information.

Example:

If you have a variable for your app name:



You can create a resource group with:

A close-up of a number

Description automatically generated

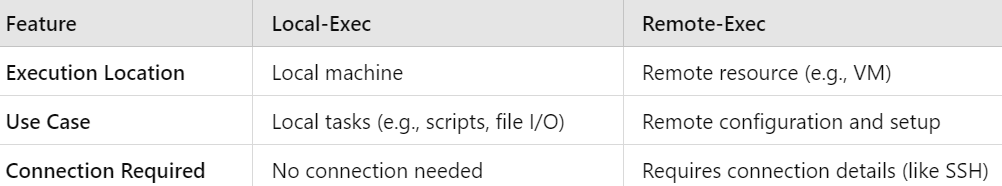
interpolation makes your Terraform code more flexible and easier to manage by letting you use dynamic values!

**Terraform Provisioner:**

In Terraform, a provisioner is a tool that allows you to execute scripts or commands on your resources after they are created. Think of it as a way to set up or configure your Azure resources automatically once they’re provisioned. Provisioners are used to run tasks like installing software, configuring settings, or copying files right after the resource is created.

**Types of Provisioners**:

* **remote-exec**: Runs commands on a remote machine, like a virtual machine in Azure.
* **local-exec**: Runs commands on your local machine, where Terraform is being executed.



 A screenshot of a computer code

Description automatically generated

In summary, use **local-exec** for tasks that need to run on your local machine and **remote-exec** for tasks that need to configure or set up resources on remote servers.