Background:

Suppose, you are working as an Infrastructure team, and they asked you to created EC2 Instance, and let’s say you are doing in manually.

Please refer: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html>

Let say this task is taking you 5 minutes, but every-time you are getting similar request then you need to think for automation.

Infrastructure as Code (IaC)

Before the advent of IaC, infrastructure management was typically a manual and time-consuming process. System administrators and operations teams had to:

1. Manually Configure Servers: Servers and other infrastructure components were often set up and configured manually, which could lead to inconsistencies and errors.

2. Lack of Version Control: Infrastructure configurations were not typically version-controlled, making it difficult to track changes or revert to previous states.

3. Documentation Heavy: Organizations relied heavily on documentation to record the steps and configurations required for different infrastructure setups. This documentation could become outdated quickly.

4. Limited Automation: Automation was limited to basic scripting, often lacking the robustness and flexibility offered by modern IaC tools.

5. Slow Provisioning: Provisioning new resources or environments was a time-consuming process that involved multiple manual steps, leading to delays in project delivery.

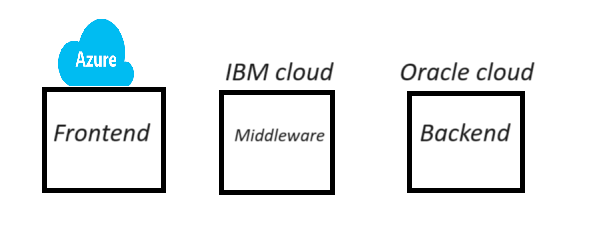
IaC addresses these challenges by providing a systematic, automated, and code-driven approach to infrastructure management. Popular IaC tools include Terraform, AWS CloudFormation, Azure Resource Manager templates others.

These tools enable organizations to define, deploy, and manage their infrastructure efficiently and consistently, making it easier to adapt to the dynamic needs of modern applications and services.

Why Terraform?

There are multiple reasons why Terraform is used over the other IaC tools but below are the main reasons.

Background: Let say I have my application as below-



Now, problem here is echo cloud have their own cloud solution, but that will not work for other.

Example: AWS has cloud formation, Azure has ARM template, so instead of using multiple things we can use terraform.

1. Multi-Cloud Support: Terraform is known for its multi-cloud support. It allows you to define infrastructure in a cloud-agnostic way, meaning you can use the same configuration code to provision resources on various cloud providers (AWS, Azure, Google Cloud, etc.) and even on-premises infrastructure. This flexibility can be beneficial if your organization uses multiple cloud providers or plans to migrate between them.

2. Large Ecosystem: Terraform has a vast ecosystem of providers and modules contributed by both HashiCorp (the company behind Terraform) and the community. This means you can find pre-built modules and configurations for a wide range of services and infrastructure components, saving you time and effort in writing custom configurations.

3. Declarative Syntax: Terraform uses a declarative syntax, allowing you to specify the desired end-state of your infrastructure. This makes it easier to understand and maintain your code compared to imperative scripting languages.

4. State Management: Terraform maintains a state file that tracks the current state of your infrastructure. This state file helps Terraform understand the differences between the desired and actual states of your infrastructure, enabling it to make informed decisions when you apply changes.

5. Plan and Apply: Terraforms "plan" and "apply" workflow allows you to preview changes before applying them. This helps prevent unexpected modifications to your infrastructure and provides an opportunity to review and approve changes before they are implemented.

6. Community Support: Terraform has a large and active user community, which means you can find answers to common questions, troubleshooting tips, and a wealth of documentation and tutorials online.

7. Integration with Other Tools: Terraform can be integrated with other DevOps and automation tools, such as Docker, Kubernetes, Ansible, and Jenkins, allowing you to create comprehensive automation pipelines.

8. HCL Language: Terraform uses HashiCorp Configuration Language (HCL), which is designed specifically for defining infrastructure. It's human-readable and expressive, making it easier for both developers and operators to work with.

To get started with Terraform, it's important to understand some key terminology and concepts. Here are some fundamental terms and explanations.

1. Provider: A provider is a plugin for Terraform that defines and manages resources for a specific cloud or infrastructure platform.

Examples of providers include AWS, Azure, Google Cloud, and many others.

You configure providers in your Terraform code to interact with the desired infrastructure platform.

2. Resource: A resource is a specific infrastructure component that you want to create and manage using Terraform. Resources can include virtual machines, databases, storage buckets, network components, and more. Each resource has a type and configuration parameters that you define in your Terraform code.

3. Module: A module is a reusable and encapsulated unit of Terraform code. Modules allow you to package infrastructure configurations, making it easier to maintain, share, and reuse them across different parts of your infrastructure. Modules can be your own creations or come from the Terraform Registry, which hosts community-contributed modules.

4. Configuration File: Terraform uses configuration files (often with a `.tf` extension) to define the desired infrastructure state. These files specify providers, resources, variables, and other settings. The primary configuration file is usually named `main.tf`, but you can use multiple configuration files as well.

5. Variable: Variables in Terraform are placeholders for values that can be passed into your configurations. They make your code more flexible and reusable by allowing you to define values outside of your code and pass them in when you apply the Terraform configuration.

6. Output: Outputs are values generated by Terraform after the infrastructure has been created or updated. Outputs are typically used to display information or provide values to other parts of your infrastructure stack.

7. State File: Terraform maintains a state file (often named `terraform.tfstate`) that keeps track of the current state of your infrastructure. This file is crucial for Terraform to understand what resources have been created and what changes need to be made during updates.

8. Plan: A Terraform plan is a preview of changes that Terraform will make to your infrastructure. When you run `terraform plan`, Terraform analyses your configuration and current state, then generates a plan detailing what actions it will take during the `apply` step.

9. Apply: The `terraform apply` command is used to execute the changes specified in the plan. It creates, updates, or destroys resources based on the Terraform configuration.

10. Workspace: Workspaces in Terraform are a way to manage multiple environments (e.g., development, staging, production) with separate configurations and state files. Workspaces help keep infrastructure configurations isolated and organized.

11. Remote Backend: A remote backend is a storage location for your Terraform state files that is not stored locally. Popular choices for remote backends include Amazon S3, Azure Blob Storage, or HashiCorp Terraform Cloud. Remote backends enhance collaboration and provide better security and reliability for your state files.

Installing terraform

Support link: <https://developer.hashicorp.com/terraform/install>

Terraform AWS Configuration: Setup Terraform for AWS

To configure AWS credentials and set up Terraform to work with AWS, you'll need to follow these steps:

1. Install AWS CLI (Command Line Interface):

Make sure you have the AWS CLI installed on your machine. You can download and install it from the [AWS CLI download page] (https://aws.amazon.com/cli/).

2. Create an AWS IAM User:

To interact with AWS programmatically, you should create an IAM (Identity and Access Management) user with appropriate permissions. Here's how to create one:

a. Log in to the AWS Management Console with an account that has administrative privileges.

b. Navigate to the IAM service.

c. Click on "Users" in the left navigation pane and then click "Add user."

- Choose a username, select "Programmatic access" as the access type, and click "Next: Permissions."

- Attach policies to this user based on your requirements. At a minimum, you should attach the "AmazonEC2FullAccess" policy for basic EC2 operations. If you need access to other AWS services, attach the relevant policies accordingly.

- Review the user's configuration and create the user. Be sure to save the Access Key ID and Secret Access Key that are displayed after user creation; you'll need these for Terraform.

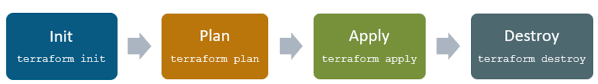
3. Configure AWS CLI Credentials:

Use the AWS CLI to configure your credentials. Open a terminal and run:

aws configure

It will prompt you to enter your AWS Access Key ID, Secret Access Key, default region, and default output format. Enter the credentials you obtained in the previous step.

Terraform lifecycle



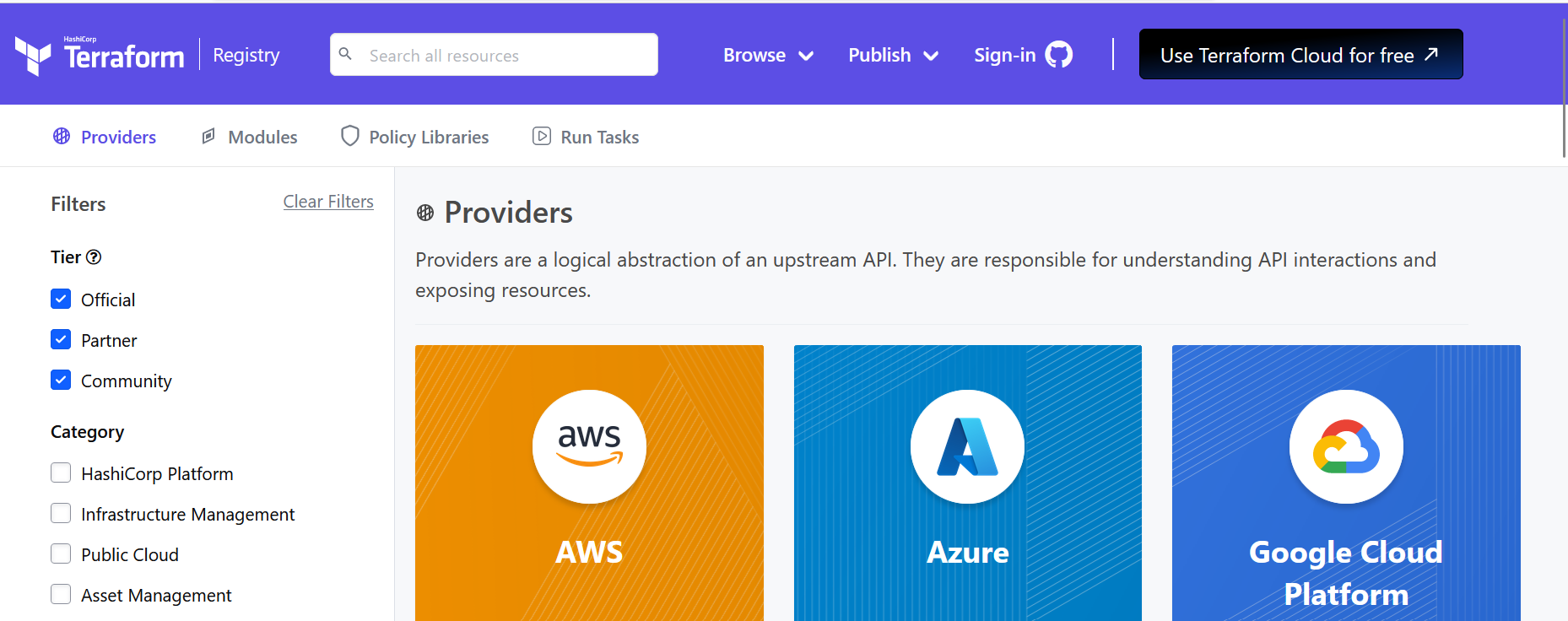
Types of Terraform providers:

Link: <https://registry.terraform.io/browse/providers>

Official: Terraform manage these providers

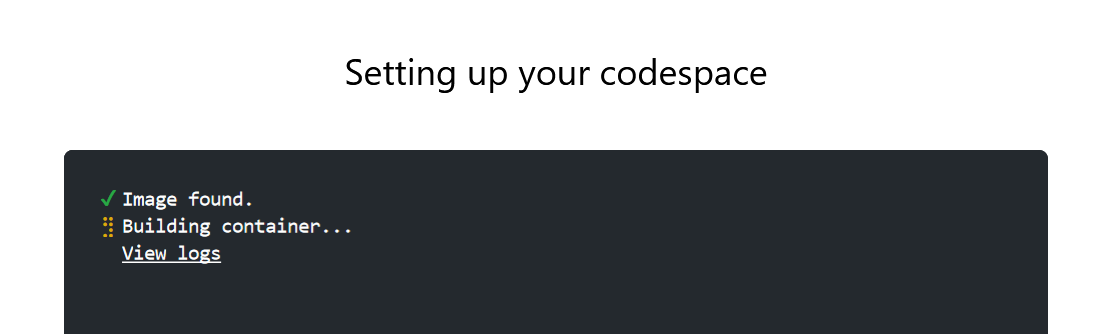
Partner: Individual company link Oracle manage their provider

Community: Group of people called community manage those providers.

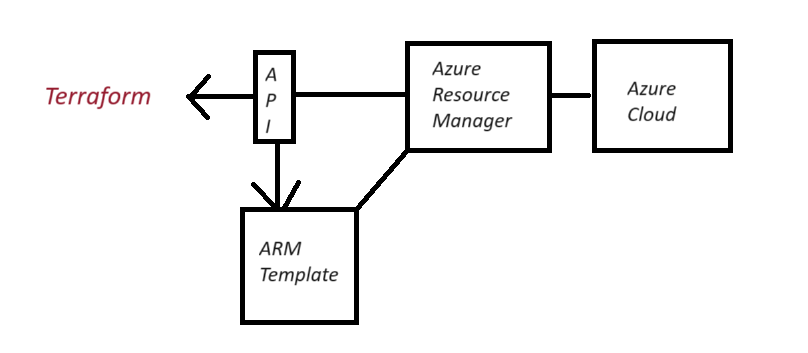


GIT Code space:

Link: <https://github.com/codespaces>



Terraform Azure Architecture:



Terraform architecture:

