What is Terraform?

- Terraform is an **open-source infrastructure as code** (IAC) tool that allows you to define and manage your cloud resources **declaratively**, using a high-level configuration language.
- Terraform is a **powerful and flexible** tool for managing cloud infrastructure that can help you streamline your workflows, reduce costs, and increase the agility and reliability of your systems.
- Terraform uses a **state file to keep track of the current state** of your infrastructure and to perform updates in a safe and efficient manner.
- It also provides a powerful set of tools for managing complex environments, including modules, workspaces, and a vast ecosystem of third-party plugins and integrations.

With Terraform:

- **Public cloud providers**: Terraform supports major public cloud providers such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and many others. You can use Terraform to create and manage virtual machines, storage, load balancers, and other cloud resources in these environments.
- **Private cloud environments**: If you have a private cloud environment based on technologies such as OpenStack, VMware, or Kubernetes, you can use Terraform to manage the infrastructure as code.
- **Hybrid cloud environments**: Terraform can also be used to manage hybrid cloud environments that span multiple cloud providers, as well as on-premises resources.
- **Networking infrastructure**: Terraform can be used to manage networking infrastructure such as virtual private clouds (VPCs), subnets, and security groups.
- **Databases**: Terraform supports popular databases such as MySQL, PostgreSQL, and MongoDB, allowing you to create and manage database instances, users, and permissions.
- **Containers**: Terraform supports container management platforms such as Kubernetes, allowing you to create and manage container clusters and related resources.

Why Terraform?

There are several reasons why you might choose to use Terraform as your infrastructure as code tool.

- **Multi-cloud support**:
    - Terraform supports *multiple cloud providers*, allowing you to manage resources across different clouds with a consistent workflow.
- **Declarative syntax**: making it easy to understand and maintain your code.
    - With this approach, you simply specify the desired state of your infrastructure, and terraform takes care of the details of how to get there.
- **Idempotent updates**: Terraform ensures that updates to your infrastructure are idempotent.
    - Meaning that applying the same configuration multiple times will always result in the same end state. This makes updates more predictable and less error-prone.
- **Infrastructure as code**: meaning that your infrastructure is treated like any other code artifact.
    - This enables you to use modern software development practices like
        - version control,
        - automated testing, and
        - continuous integration and delivery (CI/CD).
- **Resource graph**: Terraform builds a dependency graph of your resources, which allows it to perform updates in the correct order and handle dependencies automatically. This makes it easier to manage complex infrastructures with many interdependent resources.
- **Large ecosystem**: Terraform has a large and active ecosystem of providers, modules, and plugins, making it easy to extend and customize your infrastructure management workflows. This can save your time and effort by leveraging existing community-contributed resources.

How Terraform works?

HashiCorp Terraform is an Infrastructure as Code (IaC) tool that works by using a declarative language to define infrastructure resources and their dependencies. Terraform then uses this configuration to provision and manage the resources on various cloud platforms and services.

Here is a more detailed explanation of how Terraform works:

- **Configuration File**:
  - You create a configuration file in HashiCorp Configuration Language (HCL) or JSON format that defines the infrastructure resources you want to create, modify or delete.
  - The configuration file specifies the provider to use, the resources to create, and their desired state.
- **Initialization**:
  - Before you can use Terraform to manage your infrastructure, you need to initialize the working directory.
  - This is done using the "terraform init" command, which downloads the necessary provider plugins and modules.
- **Planning**:
  - Once the working directory is initialized, terraform creates a plan that shows the changes it will make to the infrastructure resources to match the desired state.
  - The plan includes the actions that will be taken, such as creating new resources, modifying existing resources, or destroying resources.
- **Execution**:
  - After reviewing the plan, you can apply the changes using the "terraform apply" command. Terraform then executes the actions required to create, modify or delete the infrastructure resources.
  - Terraform manages the dependencies between resources and ensures that they are created in the correct order.
- **State Management**:
  - Terraform stores the current state of the infrastructure resources in a state file, which is used to compare the desired state with the actual state.
  - The state file is also used to track changes over time and to identify differences between the desired and actual state.
- **Continuous Management**:
  - Terraform continuously monitors the state of the infrastructure resources and updates them as necessary to maintain the desired state.
  - Terraform can also be integrated with continuous integration and continuous deployment (CI/CD) pipelines to automate the infrastructure management process.

Terraform Basics:

Terraform is an Infrastructure as Code (IaC) tool that allows you to define and manage infrastructure resources in a declarative way. Here are some Terraform basics:

- **Configuration**: Terraform uses a declarative language called HashiCorp Configuration Language (HCL) to define the infrastructure resources you want to manage. A configuration file contains the details of the resources and their dependencies, and it describes the desired state of the infrastructure.
- **Providers**: Terraform supports multiple cloud providers, such as AWS, Azure, and Google Cloud, and each provider has a corresponding Terraform provider that allows Terraform to interact with the cloud platform. You need to specify the provider you want to use in your configuration file.
- **Resources**: Terraform manages resources like virtual machines, networks, and storage accounts. Each resource is defined by its type, such as "aws_instance" for an Amazon Web Services (AWS) instance. Resources can have attributes like size, region, and name that are specified in the configuration file.
- **Modules**: Terraform modules are reusable blocks of code that encapsulate infrastructure resources and can be shared among teams. Modules can be used to create more complex infrastructure configurations by combining multiple resources together.
- **State Management**: Terraform maintains the state of the infrastructure resources in a file. The state file is used to compare the desired state with the actual state and to make changes as necessary. Terraform automatically manages the state file, but you can also configure it to use a remote backend, like AWS S3, for storage.
- **Command-line Interface**: Terraform provides a command-line interface (CLI) that you can use to initialize, plan, apply, and destroy your infrastructure resources. The CLI also includes other commands for managing Terraform modules and providers.
- **Plan**: Before applying changes, terraform generates a plan that shows what changes will be made to the infrastructure resources. This allows you to preview changes before applying them.
- **Apply**: After reviewing the plan, you can apply the changes by running the "terraform apply" command. Terraform will create, modify, or delete the resources as necessary to match the desired state.

Terraform basic commands:

Here are some basic Terraform commands that you can use to manage your infrastructure:

- **terraform init**: This command initializes a new or existing Terraform working directory. It downloads the necessary provider plugins and sets up the backend, which is used to store the state of the infrastructure.
- **terraform plan**: This command creates an execution plan for the infrastructure changes you have defined in your Terraform configuration files. It compares the current state with the desired state and shows you what actions will be taken to reach the desired state.
- **terraform apply**: This command applies the changes defined in your Terraform configuration files to your infrastructure. It creates or modifies the necessary resources to match the desired state.
- **terraform destroy**: This command destroys all the infrastructure resources defined in your Terraform configuration files. It is useful when you want to tear down the infrastructure or start fresh.
- **terraform validate**: This command validates the syntax of your Terraform configuration files. It checks for errors and ensures that the files are valid HCL code.
- **terraform state**: This command provides various subcommands to manage the state of your infrastructure resources. You can use it to inspect the current state, move resources between state files, or import existing resources into the state file.
- **terraform output**: This command displays the values of the output variables defined in your Terraform configuration files. Output variables are used to pass information from one resource to another or to display information to the user.
- **terraform refresh**: This command updates the state file with the current state of the infrastructure resources. It is useful when the state file gets out of sync with the actual state of the resources.

Practical:

- Command 1: **terraform init** → to initialize the environment.
- Command 2: **terraform plan** → to create the plan of execution.
- Command 3: **terraform apply** → to apply the changes.
- Command 4: **terraform destroy** → to destroy the complete environment.

To save public key to id_rsa:

$ **terraform output-raw tls_private_key > id_rsa**

To connect to azure VM:

$ **ssh -i id_rsa azureuser@<IP>**