**Terraform**

**Terraform:**

Terraform is a product by Hashicorp that uses Infrastructure as Code (IaC) to provision cloud infrastructure.

It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle.

**What is Terraform?**

Terraform is an IaC tool that allows users to provision and manage infrastructure resources across various cloud platforms and on-premises environments. It uses declarative configuration files, written in HashiCorp Configuration Language (HCL) or JSON, to define and automate the lifecycle of resources, ensuring predictability and consistency. Terraform’s extensible plugin-based architecture supports a wide range of providers, enabling seamless integration and management of diverse infrastructure environments.

**What is Infrastructure as Code with Terraform?**

Infrastructure as Code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

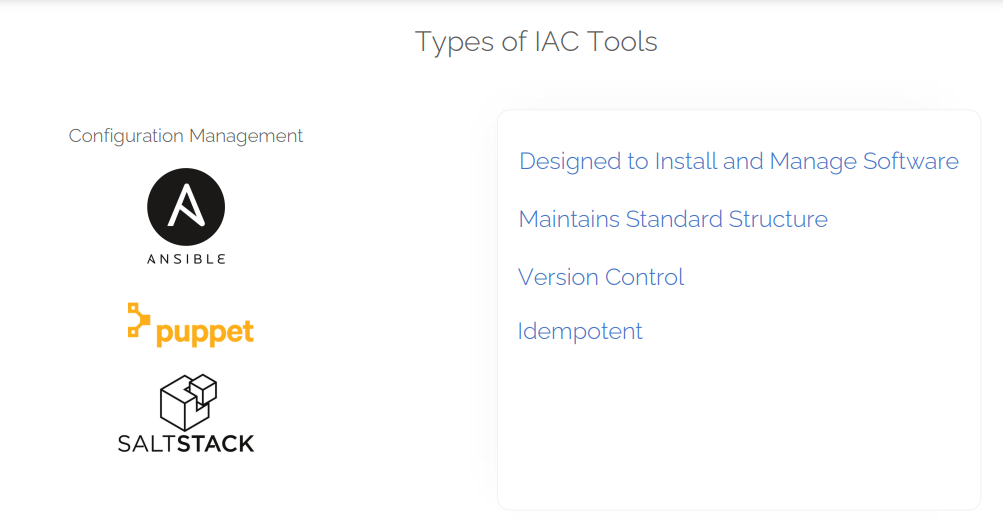
* Terraform can manage infrastructure on multiple cloud platforms.
* The human-readable configuration language helps you write infrastructure code quickly.
* Terraform's state allows you to track resource changes throughout your deployments.
* You can commit your configurations to version control to safely collaborate on infrastructure.

**Popular IaC Tools:**

1. **Terraform** An open-source declarative tool that offers pre-written modules to build and manage an infrastructure.  
   **2. Chef:** A configuration management tool that uses cookbooks and recipes to deploy the desired environment. Best used for Deploying and configuring applications using a pull-based approach.  
   **3. Puppet:** Popular tool for configuration management that follows a Client-Server Model. Puppet needs agents to be deployed on the target machines before the puppet can start managing them.  
   **4. Ansible:** Ansible is used for building infrastructure as well as deploying and configuring applications on top of them. Best used for Ad hoc analysis.  
   **5. Packer:** Unique tool that generates VM images (not running VMs) based on steps you provide. Best used for Baking compute images.  
   **6. Vagrant:** Builds VMs using a workflow. Best used for Creating pre-configured developer VMs within VirtualBox.

A screenshot of a computer

Description automatically generated

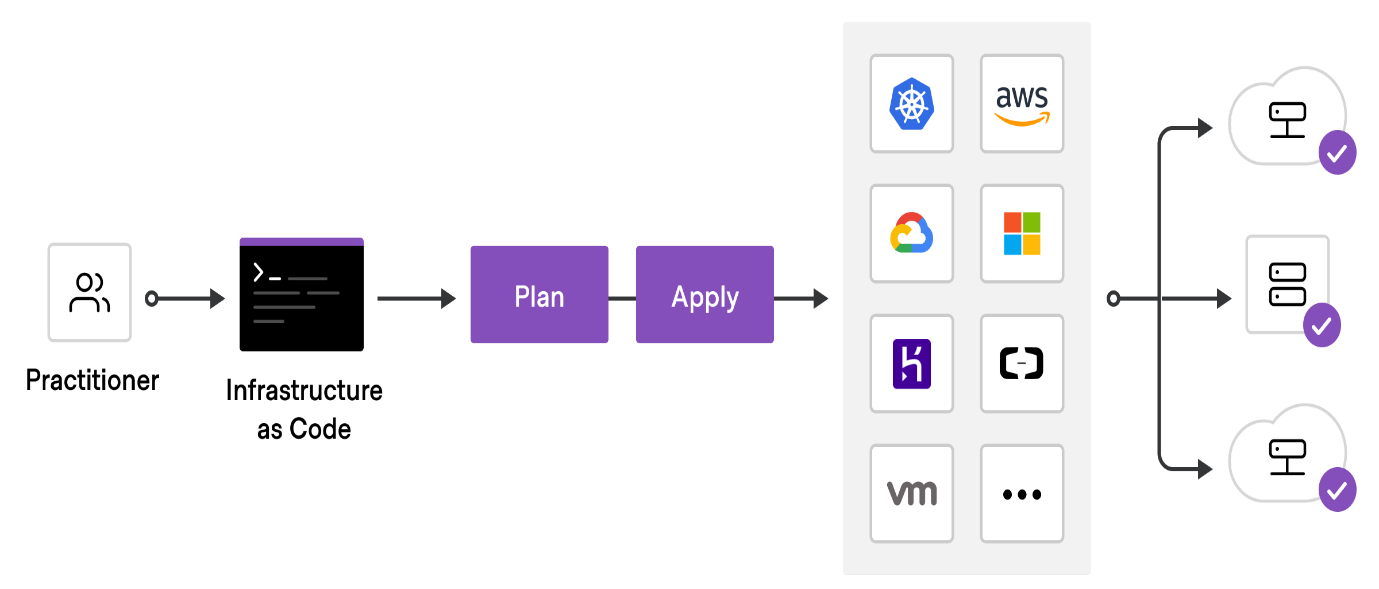


A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Terraform deployment workflow:** 

**Terraform Lifecycle:**

Terraform lifecycle consists of – init, plan, apply, and destroy.

* 1. **Terraform init** initializes the (local) Terraform environment. Usually executed only once per session.
  2. **Terraform plan**compares the Terraform state with the as-is state in the cloud, build and display an execution plan. This does not change the deployment (read-only).
  3. **Terraform apply**executes the plan. This potentially changes the deployment.
  4. **Terraform destroy**deletes all resources that are governed by this specific terraform environment.

**Terraform Core Concepts**

**1. Variables**: Terraform has input and output variables, it is a key-value pair. Input variables are used as parameters to input values at run time to customize our deployments. Output variables are return values of a terraform module that can be used by other configurations.  
Read our blog on [**Terraform Variables**](https://k21academy.com/terraform-iac/variables-in-terraform/)

**2. Provider**: Terraform users provision their infrastructure on the major cloud providers such as AWS, Azure, OCI, and others. A *provider* is a plugin that interacts with the various APIs required to create, update, and delete various resources.  
Read our blog to know more about [**Terraform Providers**](https://k21academy.com/terraform-iac/terraform-providers-overview/)

**3. Module**: Any set of Terraform configuration files in a folder is a *module*. Every Terraform configuration has at least one module, known as its ***root module.***

**4. State**: Terraform records information about what infrastructure is created in a Terraform *state* file. With the state file, Terraform is able to find the resources it created previously, supposed to manage and update them accordingly.

**5. Resources**: Cloud Providers provides various services in their offerings, they are referenced as Resources in Terraform. Terraform resources can be anything from compute instances, virtual networks to higher-level components such as DNS records. Each resource has its own attributes to define that resource.

**6. Data Source**: Data source performs a read-only operation. It allows data to be fetched or computed from resources/entities that are not defined or managed by Terraform or the current Terraform configuration.

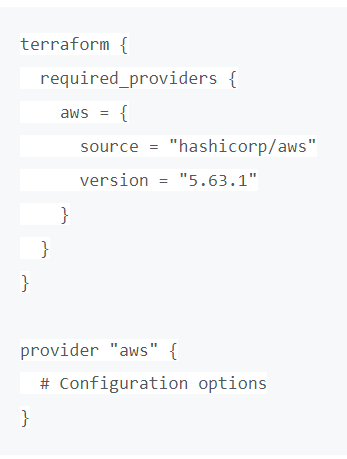
**7. Plan**: It is one of the stages in the Terraform lifecycle where it determines what needs to be created, updated, or destroyed to move from the real/current state of the infrastructure to the desired state.

**8. Apply**: It is one of the stages in the Terraform lifecycle where it applies the changes real/current state of the infrastructure in order to achieve the desired state.

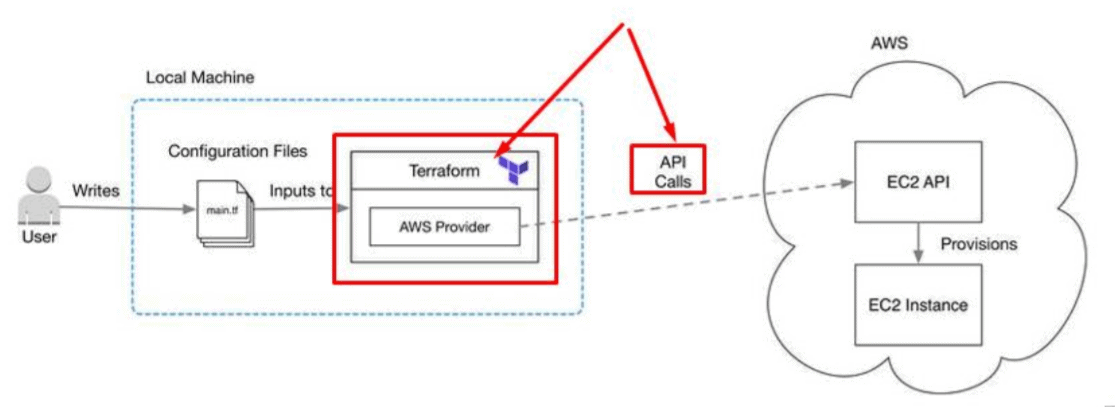
**Check Out:**Our previous blog post on [**Terraform Cheat Sheet**](https://k21academy.com/terraform-iac/terraform-cheat-sheet/).

**Terraform Providers:**

**Aws Provider**

****

A provider is responsible for understanding API interactions and exposing resources. It is an executable plug-in that contains code necessary to interact with the API of the service. Terraform configurations must declare which providers they require so that Terraform can install and use them.

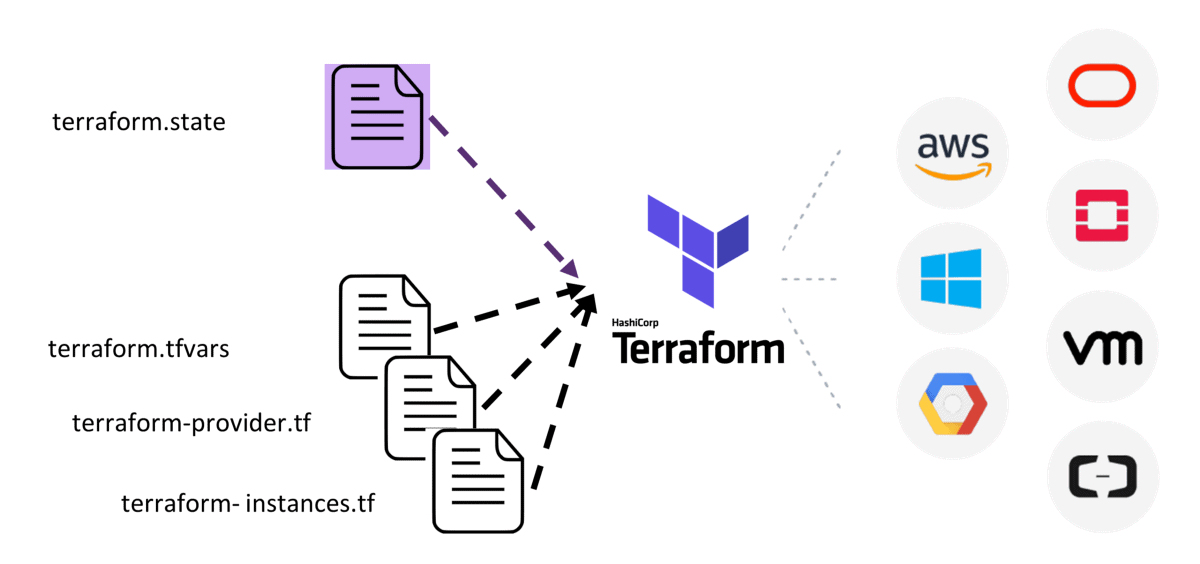


Terraform has over a hundred providers for different technologies, and each provider then gives terraform user access to its resources. So through AWS provider, for example, you have access to hundreds of AWS resources like EC2 instances, the AWS users, etc.

**Read More:**About [**Terraform Workflow**](https://k21academy.com/terraform-iac/terraform-workflow-and-its-use-case/).

**Terraform Configuration Files**

Configuration files are a set of files used to describe infrastructure in Terraform and have the file extensions **.tf** and **.tf.json**. Terraform uses a declarative model for defining infrastructure. Configuration files let you write a configuration that declares your desired state. Configuration files are made up of resources with settings and values representing the desired state of your infrastructure.



A Terraform configuration is made up of one or more files in a directory, provider binaries, plan files, and state files once Terraform has run the configuration.

**1. Configuration file (\*.tf files):** Here we declare the provider and resources to be deployed along with the type of resource and all resources specific settings

**2. Variable declaration file (variables.tf or variables.tf.json):** Here we declare the input variables required to provision resources

**3. Variable definition files (terraform.tfvars):** Here we assign values to the input variables

**4. State file (terraform.tfstate):** a state file is created once after Terraform is run. It stores state about our managed infrastructure.

**Also Read:**Our blog post on [**Terraform Create VM**](https://k21academy.com/terraform-iac/terraform-automate-aws-vm/).

For more information:  
https://k21academy.com/terraform-iac/terraform-beginners-guide/

**Terraform resources:**

**A computer screen with text

Description automatically generated**

**Terraform variables:**

Variables introduce the flexibility and dynamism needed to manage larger sets of infrastructure.

There are mainly three types of variables in Terraform – local, input, and output.

**Local variables:**

These are locally declared variables.

They provide a way to name any attribute value that needs to be used throughout the Terraform code. [Local variables](https://spacelift.io/blog/terraform-locals) are particularly useful when you need to reference the same values in multiple places within your Terraform configuration.

variables.tf main.tf

A computer screen shot of a program code

Description automatically generated

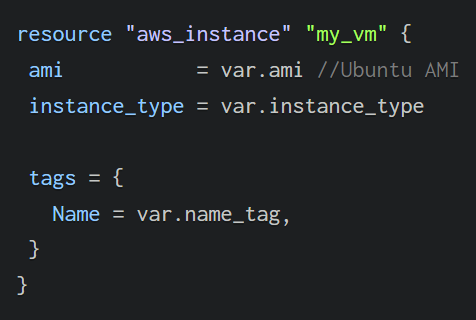
**A computer screen with text and symbols

Description automatically generated with medium confidence**

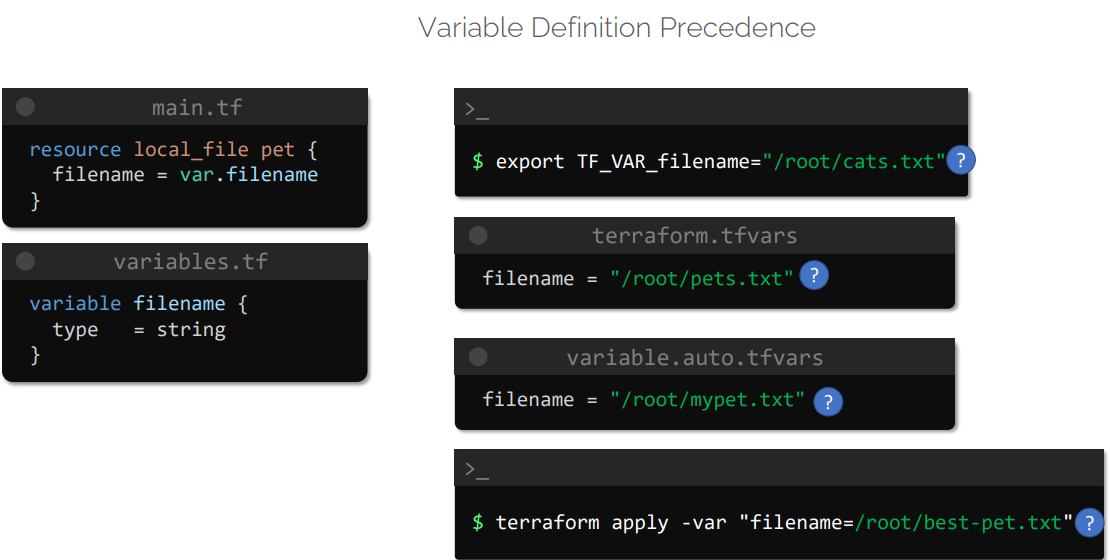
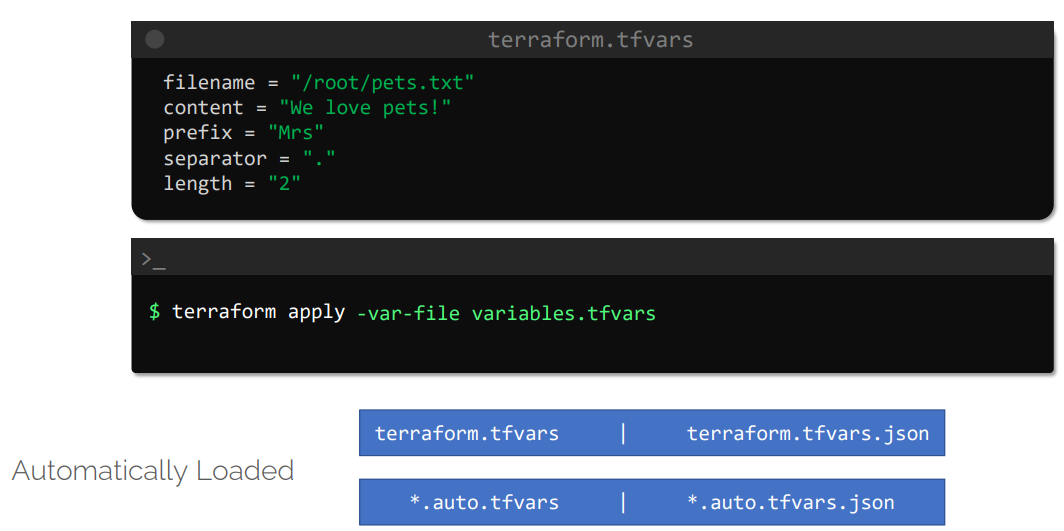
**Input variables**

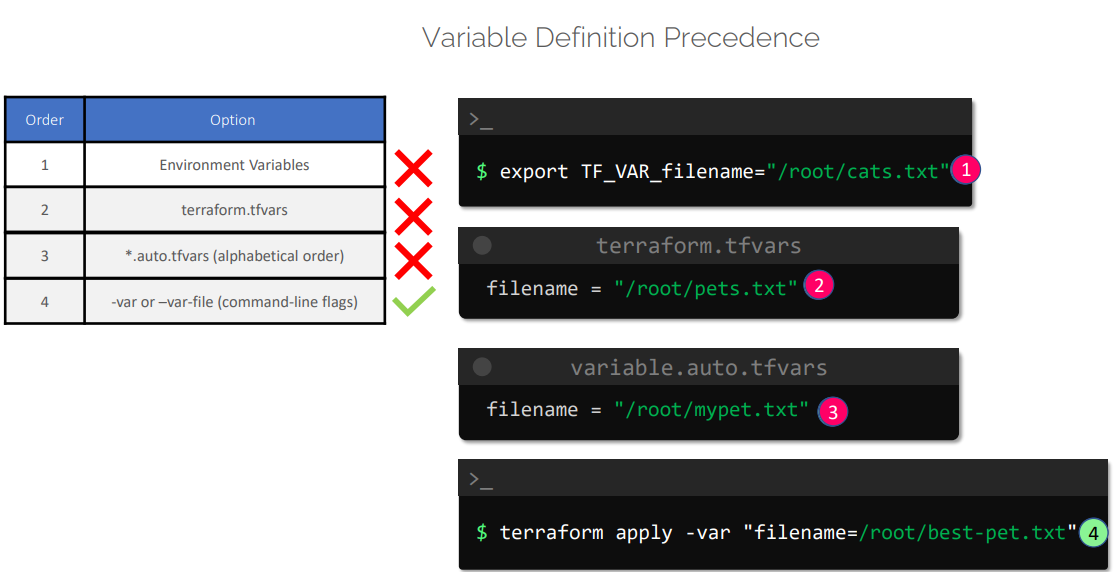
variables.tf main.tf

A screen shot of a computer program

Description automatically generated

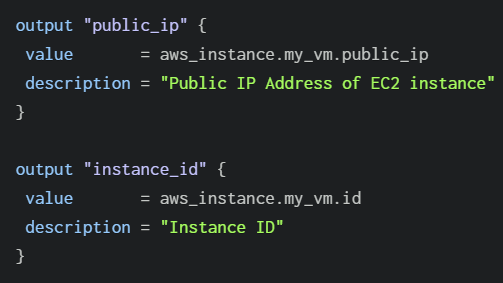
**Terraform.tfvars:**

A better way to manage these default values is to create another file named terraform.tfvars. Terraform automatically interprets tfvars as a group of key-value pairs and maps them with the declared variables in the variables.tf file. 



**Output Variables**

[Output Variables](https://spacelift.io/blog/terraform-output) provide a way to retrieve the details we are interested in directly in the CLI terminal. Additionally, output variables “return” values to the parent modules for further processing.



**Terraform Commands:**

1. **terraform validate**
2. **terraform fmt**
3. **terraform show**
4. **terraform providers**
5. **terraform output**
6. **terraform refresh**
7. **terraform graph**

**Terraform State management:**

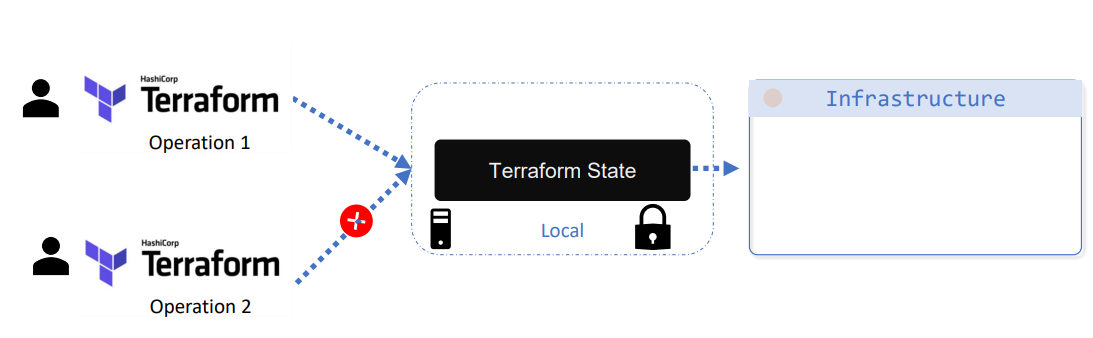
State management in Terraform is one of the crucial things to know and learn about especially when working with teams

1. terraform.tfstate
2. terraform.tfstate.backup

**State lock file**

There is one more file involved in all of this – .terraform.tfstate.lock.info.

**State Locking**

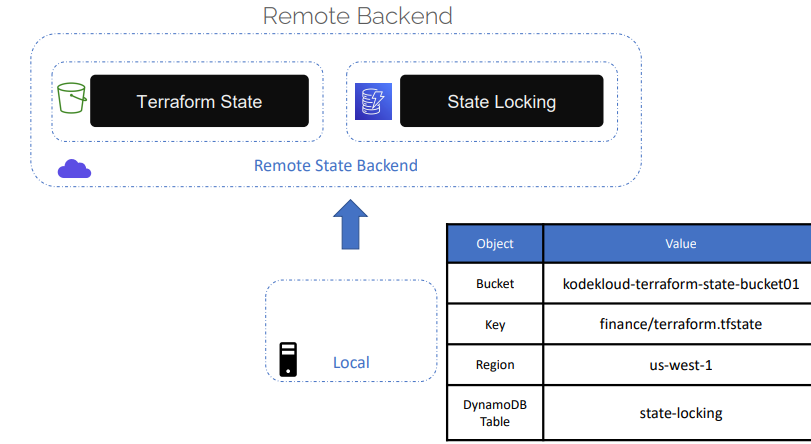


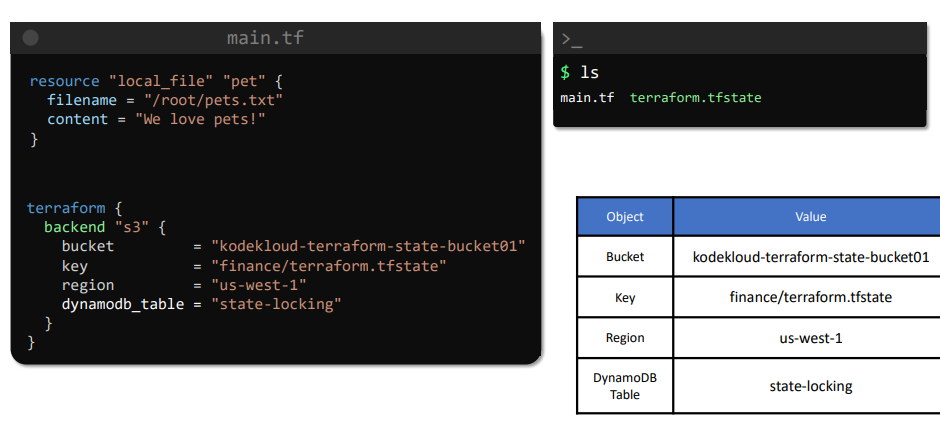
A screenshot of a computer

Description automatically generated

A screenshot of a state locking system

Description automatically generated





A screenshot of a computer program

Description automatically generated

A computer screen with text on it

Description automatically generated

**Terraform Taint**

**Debugging:**

Log Levels

**#export TF\_LOG=TRACE**

**Terraform Import**

**Data Source:**

****

**Terraform Modules:**

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

Modules are the main way to package and reuse resource configurations with Terraform.

**Terraform AWS modules**

<https://registry.terraform.io/namespaces/terraform-aws-modules>

<https://github.com/terraform-aws-modules>

**Terraform Functions:**

|  |  |
| --- | --- |
| **Function Type** | **Description** |
| **String** | **String related operations (format, join, split)** |
| **Numeric** | **Numeric related operations (min, max, pow)** |
| **Collection** | **Functions that manipulate lists, tuples, sets and maps (length, lookup, merge)** |
| **Date and Time** | **Manipulate date and time (formatdate, timestamp)** |
| **Crypto and Hash** | **Crypto and Hash functions (base64sha512, bcrypt)** |
| **Filesystem** | **File system operations (file, filexists, abspath)** |
| **Ip Network** | **Network Cidr functions (cidrsubnet, cidrhost)** |
| **Encoding** | **Encoding and decoding functions (base64decode, base64encode, jsonencode)** |
| **Type Conversion** | **Functions that convert data types (tobool, tomap, tolist)** |

Please follow the below link for more details.

<https://spacelift.io/blog/terraform-functions-expressions-loops>

<https://developer.hashicorp.com/terraform/language/functions>

**Terraform Workspaces:**

****

Refer the below link for more information:

<https://spacelift.io/blog/terraform-workspaces>

**Terraform Project**

**1.  Terraform Project with Aws resources.**

Q. Create AWS EC2 instance by supplying arguments with variables and use data source to grab the AMI (Latest ami of amazon linux) from the aws. Access the html file in the shell scripting and run  it( add data file to access shell script) and also add security groups with inbound rules as http?

A.

Q. Create S3 buckets with policy and enable the static website host and enable versioning.

Q. Create VPC using modules.

    Add 2 availability zone and 2 public and private subnets and create database subnets

    Attach internet and nat gateway and also use tfvars file for giving the variable?

Q. Create RDS and use null resources and also use locals and use environment variables?

**Docker**

Docker is containerization technology tool.

Kubernetes is container orchestration tool.

**Docker is a container runtime.**

**Containerization:**  
 Its all about deploy application with required dependencies is known as containerization.

**Kubernetes:**

Kubernetes is a platform for running and managing container for many(100’s of) containers runtimes.

**Virtualization:**

-------------------------------------------------------------------------------------------------------------------

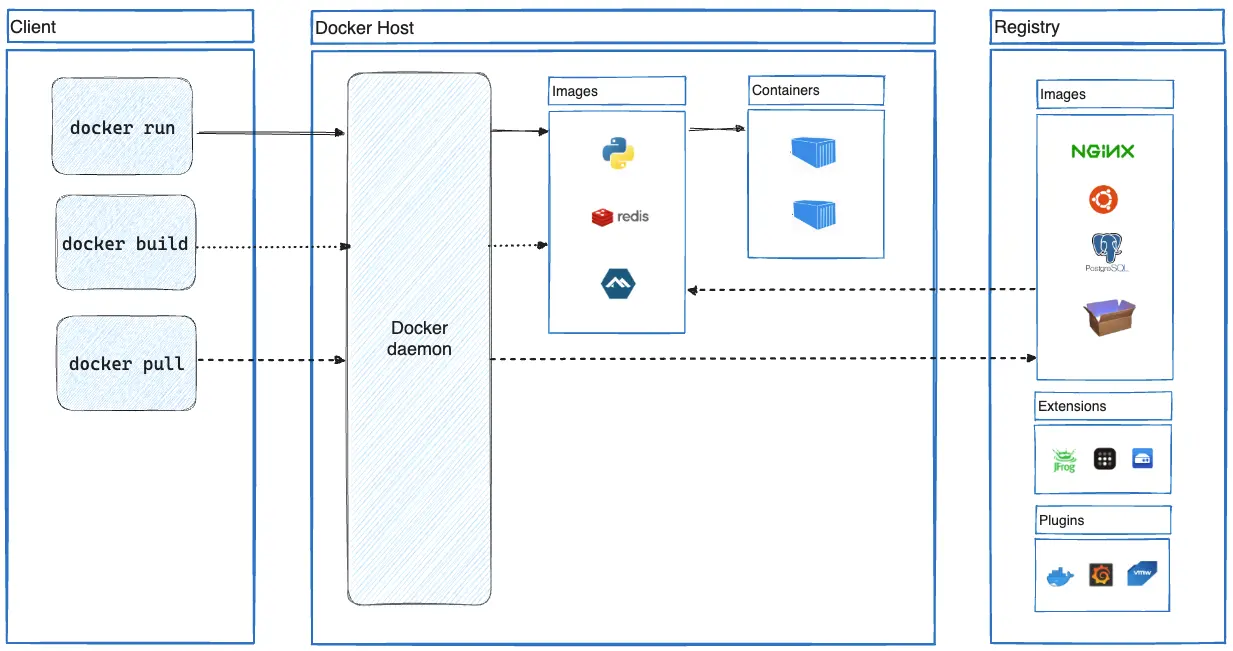
**Container Engine:**

Software which helps to implement containerization on a machine or server is caller container Engine.

Tool:

Docker, Jail, Crio.

**Docker Architecture:**



**Docker CLI/Client:**

Docker client is used to interact with docker hub to interact with images or containers.

* **Docker pull :** It helps to get the images from docker hub into the server.
* **Docker build:** To create docker images.
* **Docker run:** To create the containers from the images.

**Docker Daemon (Docker service):**

Docker daemon manages all the services by communicating with other daemons. It manages docker objects such as images, containers, networks, and volumes with the help of the API requests of Docker.

(The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. )

**Docker Host**

A Docker host is a type of machine that is responsible for running more than one container. It comprises the Docker daemon, Images, Containers, Networks, and Storage.

**Docker Registry**

All the docker images are stored in the docker registry.

**Registry** : It is a collection of all dependencies and docker container images

**Docker pull:** It helps to download the images or dependencies from registry on to the docker host.

Docker pull <image name:version>

**Docker Images**

An image contains instructions for creating a docker container. It helps to create container from docker images and it is used to store and ship applications.

**Docker Containers**

Containers are created from docker images. With the help of Docker API or CLI, we can start, stop, delete, or move a container.

**Docker Services** :

* After docker install, we have to follow 3 steps (or) activities.

1. Start the docker services.

Ex: service docker start

1. Enable docker services at boot time.

chkconfig docker on

1. Add the user account to the docker root group.

Ex: usermod -a -G dockerroot <username>

* After installing docker, it will create group called dockerroot. We must add user account to the docker root group.

**Docker commands:**

How to check docker client and engine version ?  
docker version

How to check docker server configuration ?

docker info

How to find the container image ?

Docker images

How to check the container information?

Docker ps -a

Or

Docker ps -> it will shows only live or running containers

How to create images ?

Docker build –tag <image Name> <file path to docker file>

How to create a container ?

Docker run -it <image name or Id> <container shell ex: sh, bash>

Note : -it is used for create and log into the container

How to close a container and come out ?

Type exit in container and click on enter.

How to come out from container without closing session ?

Ctrl + pq

How to stop container ?

Docker container stop <container id or name>

How to start container ?

Docker container start <container id or name>

How to log into the container or docker exec command ?

Docker exec -it <container name or id > <container shell>

Note : the container should be running and up to log into the container

If container is stopped or not running we have to start and log in.

How to start and log into a container?

Docker start -ai <container name or id >

Note : Here we can’t change the shell, default shell is sh

Create or update container names ?

Docker rename <old container name > <new container name>

How to name a container while creating ?

Docker run –name <container name> -it <image id> <container shell>

How to delete containers ?

Docker rm <container id or name>

Note: always delete a container when status is showing as exited.

docker rm <container id or name> --force ( to remove container forcefully)

How to delete multiple containers ?

Docker rm < container name1> name2 name3 etc…

How to delete docker images ?

Docker rmi <image name or id >

Note : it will only delete images not the container running from those images.

Image tagging :

Two types of image tags are available.

1. Local tag

Local tag helps to create an alias name use case on docker server or host. We can’t upload local tag images to docker registry

Docker image tag <source image name or id > <new image name >

1. Remote tag

We can generate image alias both for use case or keeping images on docker host and upload to the docker registry.

Docker tag <source image name or id > <docker hub id>/<new image name>:<image version>

How to login to docker hub from cli mode ?

Docker login

How to push images to docker registry or hub ?

Docker push <image name>

Note: **only use image names to push images to docker hub**

How to logout from docker hub ?

Docker logout

Docker image inspect:

Inspect command shows the properties of the image.

How to get the image meta data info or properties ?

Docker image inspect <image id or name>

How to get the container meta data info or properties ?

Docker container inspect <container name or id>

Docker image history :

This command helps to check the layers of the image or code of the image.

Docker image history

Docker layer:

Docker build follows the process of interpretation it will check the code line by line.

If we have 4 lines in a code

First line code is correct it will create hashtag or commit id. If the hashtag crated this will generate a layer.

Image is a collection of layers

In dockerfile output layers are arranged in descending order

The last line of the layer in an image automatically considered as image id.

Each line of code in dockerfile we have to call as instruction.

If instruction executed correctly docker will create layer, all these instructions are written in a file is called dockerfile. For docker file no extension.

Docker build is process of converting instructions into layers if all layers are successfully executed then it will generate a docker image else if one of the instructions fails then layer will not generate.

If one of the layer fails then docker image fails to generate.

**Dockerfile** :

* Dockerfile helps to create images,
* File contains instructions.
* These instructions more look like Linux commands.

Work flow of docker image process:

1. Crete project folder
2. Cd into folder, create dockerfile. Name is Dockerfile.
3. Open the dockerfile, write the code and save it.

Eg: FROM python:3.12

WORKDIR /usr/local/app

# Install the application dependencies

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code

COPY src ./src

EXPOSE 5000

# Setup an app user so the container doesn't run as the root user

RUN useradd app

USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]

1. Excute it (Image build process)

Commad:

Docker image build –tag <image\_name > <docker file path>

1. Result: docker ps
2. Then push to docker hub.

**Docker instructions:**

* **FROM <image>** - this specifies the base image that the build will extend.
* **WORKDIR <path>** - this instruction specifies the "working directory" or the path in the image where files will be copied and commands will be executed.
* **COPY <host-path> <image-path>** - this instruction tells the builder to copy files from the host and put them into the container image.
* **RUN <command>** - this instruction tells the builder to run the specified command.
* **ENV <name> <value>** - this instruction sets an environment variable that a running container will use.
* **EXPOSE <port-number> -** this instruction sets configuration on the image that indicates a port the image would like to expose.
* **USER <user-or-uid>** - this instruction sets the default user for all subsequent instructions.
* **CMD ["<command>", "<arg1>"]** - this instruction sets the default command a container using this image will run.

For more information about docker installations:

<https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/>

**Docker Networking**

Docker Networking allows you to create a Network of Docker Containers managed by a master node called the manager.   
A network is a group of two or more devices that can communicate with each other either physically or virtually.

The Docker network is a virtual network created by Docker to enable communication between Docker containers.

If two containers are running on the same host they can communicate with each other without the need for ports to be exposed to the host machine.

**Network Drivers:**

There are several default network drivers available in Docker.

Command to check docker networks list.

**docker network ls**

**Types of Network Drivers:**

**bridge:** If you build a container without specifying the kind of driver, the container will only be created in the bridge network, which is the default network.

**host:** Containers will not have any IP address they will be directly created in the system network which will remove isolation between the docker host and containers.

**none:** IP addresses won’t be assigned to containers. These containments are not accessible to us from the outside or from any other container.

**overlay:** overlay network will enable the connection between multiple Docker demons and make different Docker swarm services communicate with each other.

**ipvlan:** Users have complete control over both IPv4 and IPv6 addressing by using the IPvlan driver.

**macvlan:** macvlan driver makes it possible to assign MAC addresses to a container.

Network Drivers

The Docker Network command is the main command that would allow you to create, manage, and configure your Docker Network. Let’s see what the sub-commands can be used with the Docker Network command. to know more about Creating a Network in Docker and Connecting a Container to That Network.

**sudo docker network**

**How to create a docker network.**

**sudo docker network create --driver <driver-name> <bridge-name>**

Using the “Connect” command, you can connect a running Docker Container to an existing Network.

**sudo docker network connect <network-name> <container-name or id>**

Using the Network Inspect command, you can find out the details of a Docker Network.

You can also find the list of Containers that are connected to the Network.

**sudo docker network inspect <network-name>**

The disconnect command can be used to remove a Container from the Network.

**sudo docker network disconnect <network-name> <container-name>**

You can remove a Docker Network using the rm command.  
Note that if you want to remove a network, you need to make sure that no container is currently referencing the network.

**sudo docker network rm <network-name>**

To remove all the unused Docker Networks, you can use the prune command.

**sudo docker network prune**