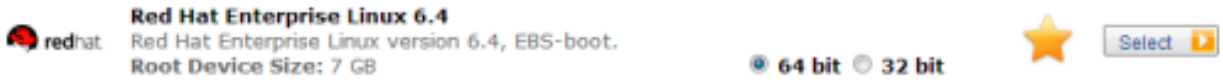
AWS EC2- Virtual machine in AWS is called EC2.

EC2 uses key based authentication mechanism. They share private key and keep public key with them. One key handshake happens successfully, then we will get active session.

Why Amazon uses key based authentication? **(AWS hold public key, and end user hold private)**
Answer: It considered as safer option for communication.

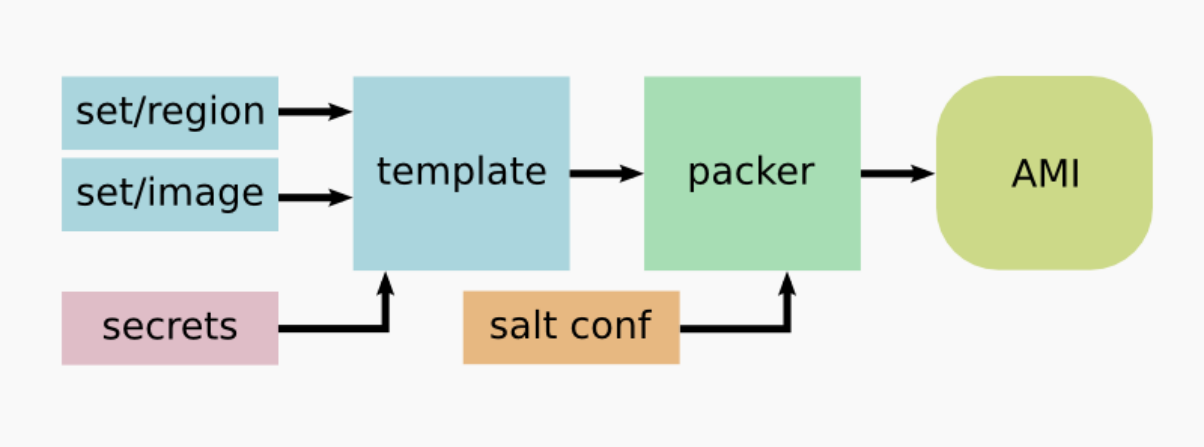
Note- Amazon uses reusable images for creating the virtual box.



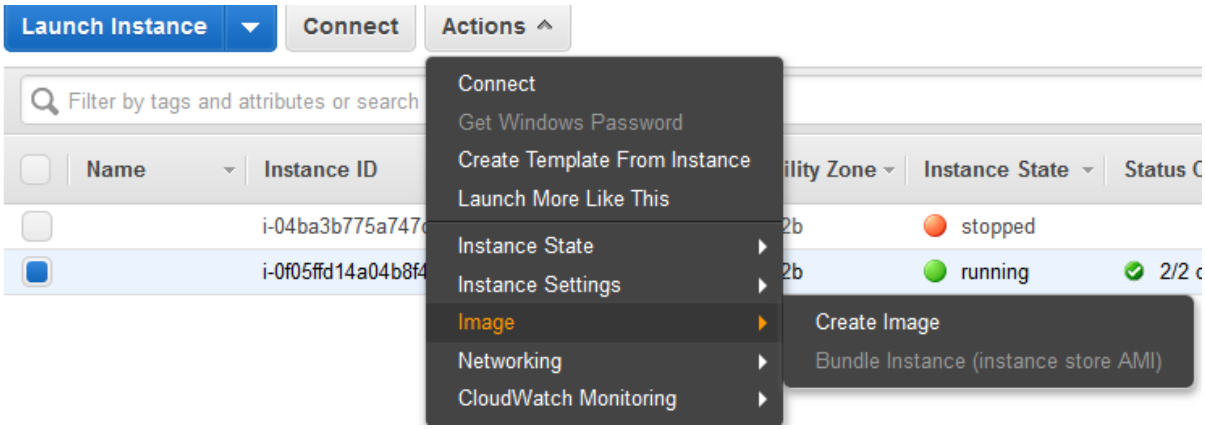
My requirement here: I want to create reusable image

Case study- There is a tool that has business intelligent in it and depending on user input it create reusable image.

Example- Packer.



Manual way to create image:



Note- For creating image, we provide JSON file as input to Packer.

Packer has 2 important components-

1) Builder
2) Provisioning.
3) Postprocessing

Builders Builders are responsible for creating machines and generating images from them for various platforms. For example, there are separate builders for EC2, VMware, VirtualBox, etc. Packer comes with many builders by default and can also be extended to add new builders.
Provisioning: It's a custom work that need to perform, like installing java on box.
Postprocessing: Once your work is done, you can change the format also.

Immutable infrastructure

An *immutable infrastructure* is another infrastructure paradigm in which servers are never modified after they're deployed. If something needs to be updated, fixed, or modified in any way, new servers built from a common image with the appropriate changes are provisioned to replace the old ones. After they're validated, they're put into use and the old ones are decommissioned.

Example- let say you have setup infrastructure with Java 7 and tomcat, and now you want to modify it to java8 and tomcat 8, so you will be setting up a new environment, and once setup is completely up and running then you will be decom previous infrastructure.

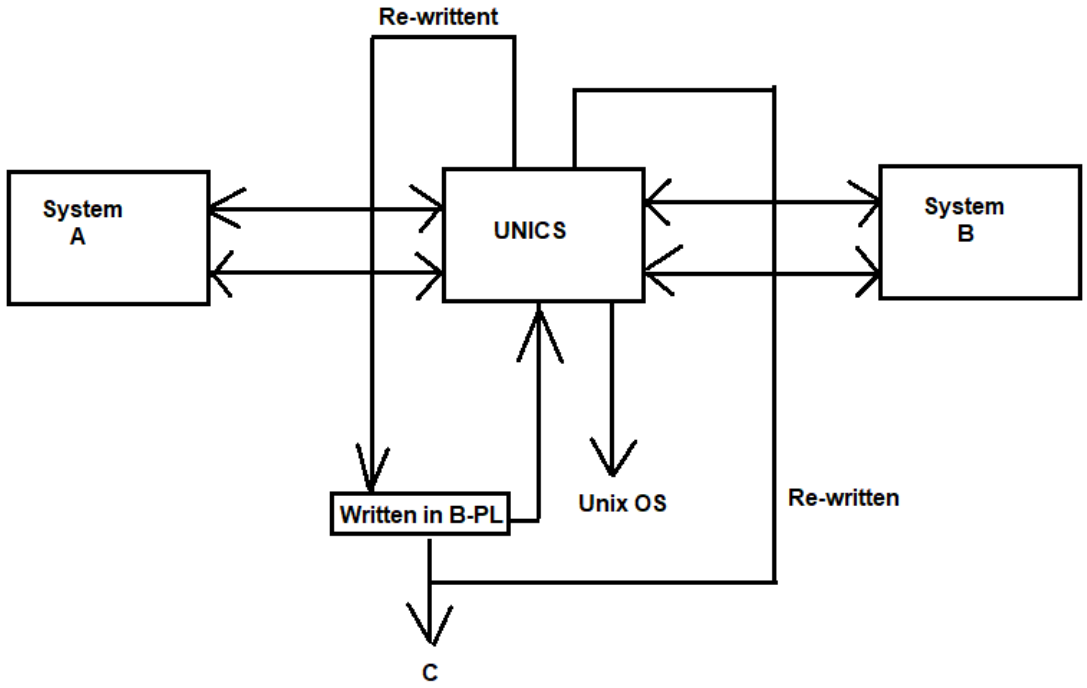
Difference between IT infra and Devops?
In IT infra we will go for troubleshooting whereas in Devops infra we will create immutable infra.

In which language packer is developed?
Go (incorrectly known as Golang, is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing.

Packer template

```
{
  "variables": {
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "AKINOPEIOMITTEDLHA",
    "secret_key": "ASUn0PeL+tU9hmOMITTEDStiLLOmittedYu",
    "region": "us-east-1",
    "source_ami": "ami-c481fad3",
    "iam_instance_profile": "S3-Admins",
    "instance_type": "t2.micro",
    "ssh_username": "ec2-user",
    "ami_name": "OrgWebServer{{timestamp}}"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sleep 30",
      "sudo yum install httpd -y",
      "sudo yum update -y",
      "sudo aws s3 cp s3://Org-webbucket/index.html /var/www/html"
      "sudo aws s3 cp s3://Org-webbucket/image001.png /var/www/htm
      "sudo service httpd start",
      "sudo chkconfig httpd on"]
  }]
}
```

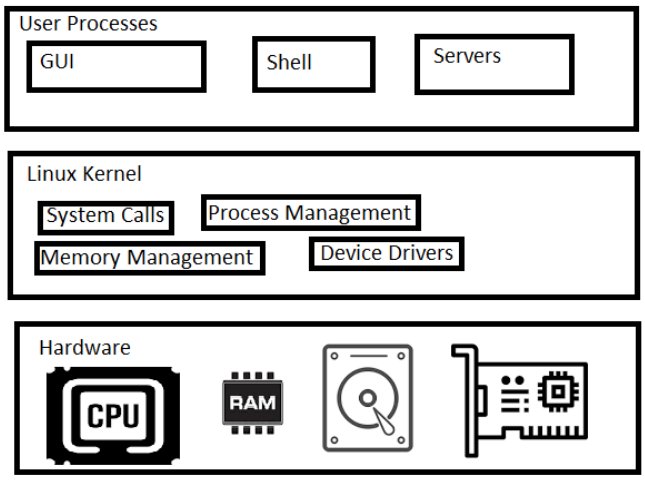
History of IT System Evolution for UNIX



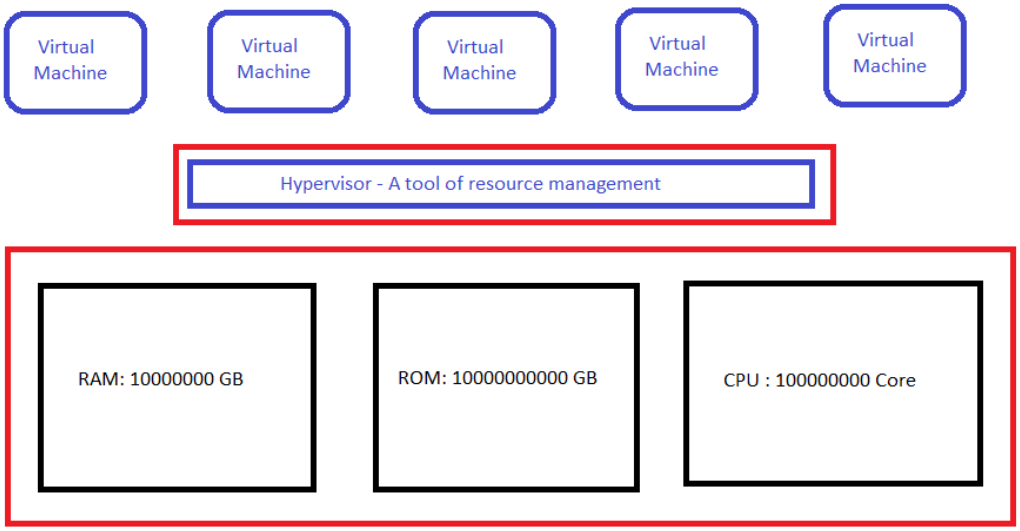
Linux= UNIX + (Utilities)

	Unix	Linux
1	It is an operating system which can only be used by it's copyrighters (IBM, Sun etc).	It is open Source and available for free download.
2	It was developed for servers and mainframes.	It was developed for home users , developers ,Students etc.
3	It supports file system but lesser than Linux.	Supports more file system than unix .
4	Developed as a project by Dennis Ritchie and Ken Thompson.	Developed by Linus Torvalds.
5	Unix is a complete package of Operating System.	Linux is a kernel . On this many systems are built like - Ubuntu,RHEL,Mint,Fedora etc.

Linux Architecture:



IT company Resource management



((Organisation infrastucture))

Case study- Let's assume a requirement where you need to deploy application, give application URL to testing team.

Application URL- http(s)://host:port/context-root

Here two cases are possible-
Case 1- You are having a Box. (requirement- Deploy your application-> provisioning)
Case 2- You are not having any box.(in this case I need to create infra then provisioning)

Creating infra-> Cloud formation, 2) Provisioning -> Ansible/chef

Selecting between Traditional Vs cloud

Conventional	Cloud
• Manually Provisioned	• Self-provisioned
• Dedicated Hardware	• Shared Hardware
• Fixed Capacity	• Elastic Capacity
• Pay for Capacity	• Pay for Use
• Capital & Operational Expenses	• Operational Expenses

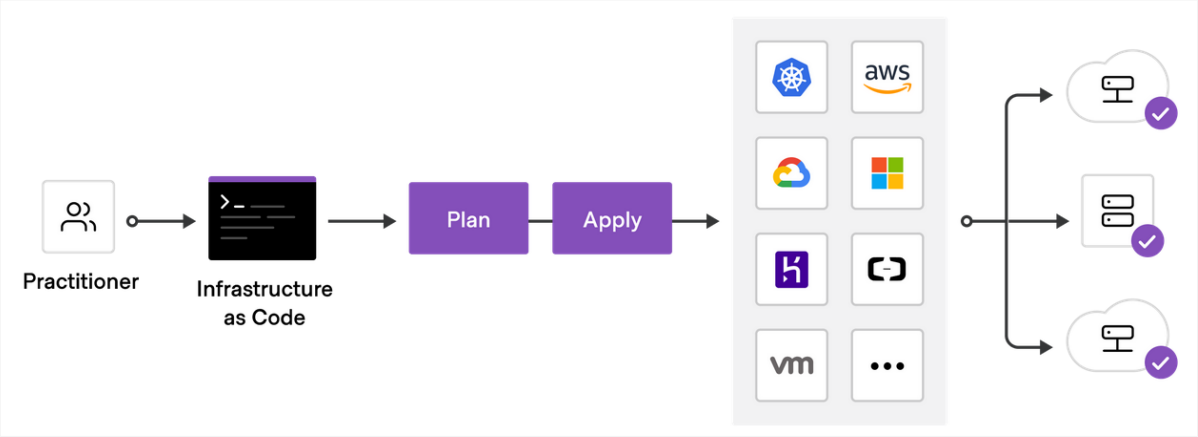
Case study- EC2 machine creation
OS- AMI id > Capacity- t2.micro >Security group >key value pair

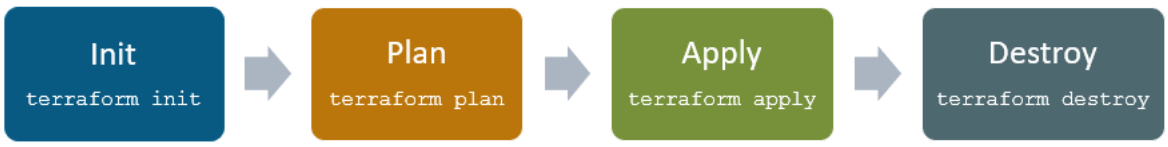
There are two ways to manage/create infrastructure:
1) By GUI (Example- AWS Console)
2) By IAC [Infrastructure as code] (Example-Terraform)Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:

- 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.

Terraform workflow:





Installing Terraform:

curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main"
sudo apt-get update && sudo apt-get install terraform

Running Terraform on AWS:

1) Create IAM User, and notedown key

Access Key: AKIA3QYLLWU44TUED5P3
Secret Key: 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

2) Create EC2 Instance, and Install terraform and packer

3) AWS CLI install:

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install

Ref:https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

4) Create main.tf file and write desire configuration there.

Write configuration

The set of files used to describe infrastructure in Terraform is known as a Terraform configuration. You will write your first configuration to define a single AWS EC2 instance.

Each Terraform configuration must be in its own working directory. Create a directory for your configuration.

$ mkdir learn-terraform-aws-instance ; cd learn-terraform-aws-instance

Create a file to define your infrastructure. $ touch main.tf

Open main.tf in your text editor, paste in the configuration below, and save the file.

Tip: The AMI ID used in this configuration is specific to the us-west-2 region. If you would like to use a different region, then write AMI accordingly.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }
  }
  required_version = ">= 0.14.9"
}
provider "aws" {
  profile = "default"
  region  = "us-west-2"
}
resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
```

```
  tags = {
    Name = "ExampleAppServerInstance"
  }
```

This is a complete configuration that you can deploy with Terraform. The following sections review each block of this configuration in more detail.

Terraform Block

The terraform {} block contains Terraform settings, including the required providers Terraform will use to provision your infrastructure. For each provider, the source attribute defines an optional hostname, a namespace, and the provider type. Terraform installs providers from the Terraform Registry by default. In this example configuration, the aws provider's source is defined as hashicorp/aws, which is shorthand for registry.terraform.io/hashicorp/aws.

Providers – Example AWS

The provider block configures the specified provider, in this case aws. A provider is a plugin that Terraform uses to create and manage your resources.

The profile attribute in the aws provider block refers Terraform to the AWS credentials stored in your AWS configuration file, which you created when you configured the AWS CLI. Never hard-code credentials or other secrets in your Terraform configuration files. Like other types of code, you may share and manage your Terraform configuration files using source control, so hard-coding secret values can expose them to attackers.

You can use multiple provider blocks in your Terraform configuration to manage resources from different providers. You can even use different providers together. For example, you could pass the IP address of your AWS EC2 instance to a monitoring resource from DataDog.

Resources – Example EC2/S3

Use resource blocks to define components of your infrastructure. A resource might be a physical or virtual component such as an EC2 instance, or it can be a logical resource such as a Heroku application.

Resource blocks have two strings before the block: the resource type and the resource name. In this example, the resource type is aws_instance and the name is app_server. The prefix of the type maps to the name of the provider. In the example configuration, Terraform manages the aws_instance resource with the aws provider. Together, the resource type and resource name form a unique ID for the resource. For example, the ID for your EC2 instance is aws_instance.app_server.

Initialize the directory

When you create a new configuration — or check out an existing configuration from version control — you need to initialize the directory with terraform init.

Initializing a configuration directory downloads and installs the providers defined in the configuration, which in this case is the aws provider.

Initialize the directory.

$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 3.27"...
- Installing hashicorp/aws v3.27.0...
- Installed hashicorp/aws v3.27.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider selections it made above. Include this file in your version control repository so that Terraform can guarantee to make the same selections by default when you run "terraform init" in the future.

Terraform has been successfully initialized!

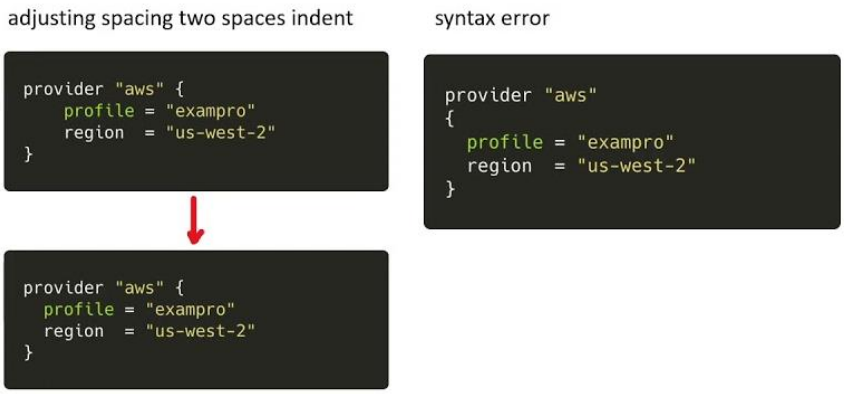
Format and validate the configuration

We recommend using consistent formatting in all of your configuration files. The terraform fmt command automatically updates configurations in the current directory for readability and consistency.

Format your configuration. Terraform will print out the names of the files it modified, if any. In this case, your configuration file was already formatted correctly, so Terraform won't return any file names.

$ terraform fmt

terraform fmt will by default look in the current directory and apply formatting to all .tf files.



You can also make sure your configuration is syntactically valid and internally consistent by using the terraform validate command.

Validate your configuration. The example configuration provided above is valid, so Terraform will return a success message.

$ terraform validate
Success! The configuration is valid.

Create infrastructure

Apply the configuration now with the terraform apply command. Terraform will print output similar to what is shown below. We have truncated some of the output to save space.

$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app_server will be created
  + resource "aws_instance" "app_server" {
      + ami   = "ami-830c94e3"
      + arn   = (known after apply)
##...

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:

Before it applies any changes, Terraform prints out the execution plan which describes the actions Terraform will take in order to change your infrastructure to match the configuration.

The output format is similar to the diff format generated by tools such as Git. The output has a + next to aws_instance.app_server, meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is (known after apply), it means that the value will not be known until the resource is created. For example, AWS assigns Amazon Resource Names (ARNs) to instances upon creation, so Terraform cannot know the value of the arn attribute until you apply the change and the AWS provider returns that value from the AWS API.

Terraform will now pause and wait for your approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure.

In this case the plan is acceptable, so type yes at the confirmation prompt to proceed. Executing the plan will take a few minutes since Terraform waits for the EC2 instance to become available.

  Enter a value: yes

aws_instance.app_server: Creating...
aws_instance.app_server: Still creating... [10s elapsed]
aws_instance.app_server: Still creating... [20s elapsed]

```
aws_instance.app_server: Still creating... [30s elapsed]
aws_instance.app_server: Creation complete after 36s [id=i-01e03375ba238b384]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Inspect state

When you applied your configuration, Terraform wrote data into a file called terraform.tfstate. Terraform stores the IDs and properties of the resources it manages in this file, so that it can update or destroy those resources going forward.

The Terraform state file is the only way Terraform can track which resources it manages, and often contains sensitive information, so you must store your state file securely and restrict access to only trusted team members who need to manage your infrastructure. In production, we recommend storing your state remotely with Terraform Cloud or Terraform Enterprise. Terraform also supports several other remote backends you can use to store and manage your state.

Inspect the current state using terraform show.

```
$ terraform show
# aws_instance.app_server:
resource "aws_instance" "app_server" {
    ami                          = "ami-830c94e3"
    arn                          = "arn:aws:ec2:us-west-2:561656980159:instance/i-01e03375ba238b384"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2c"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    hibernation                  = false
    id                           = "i-01e03375ba238b384"
    instance_state               = "running"
    instance_type                = "t2.micro"
    ipv6_address_count           = 0
    ipv6_addresses               = []
    monitoring                   = false
    primary_network_interface_id = "eni-068d850de6a4321b7"
    private_dns                  = "ip-172-31-0-139.us-west-2.compute.internal"
    private_ip                   = "172.31.0.139"
    public_dns                   = "ec2-18-237-201-188.us-west-2.compute.amazonaws.com"
    public_ip                    = "18.237.201.188"
    "Name" = "ExampleAppServerInstance"
```

When Terraform created this EC2 instance, it also gathered the resource's metadata from the from the AWS provider and wrote the metadata to the state file. Later in this collection, you will modify your configuration to reference these values to configure other resources and output values.

Manually Managing State

Terraform has a built-in command called terraform state for advanced state management. Use the list subcommand to list of the resources in your project's state.
$ terraform state list
aws_instance.app_server

Terraform allows you to create and provision application in cloud.

packer and terraform-

in packer we are not login into the boxes, but in terraform we do.(packer create AMI ID, but terraform create environment)

As you are giving directory as input, so you can define multiple tf file as input like-

1 file for input
1 for output
1 for variable

Example- variable.tf , Output.tf and main.tf

teraform init- command to initialize terraform, and depending on your input, it creates plugin.

tfvars- for defining variable in a new file

Module- like code in wora format, and share with different team like a module.

Module- they are reusable component.

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Common command options:

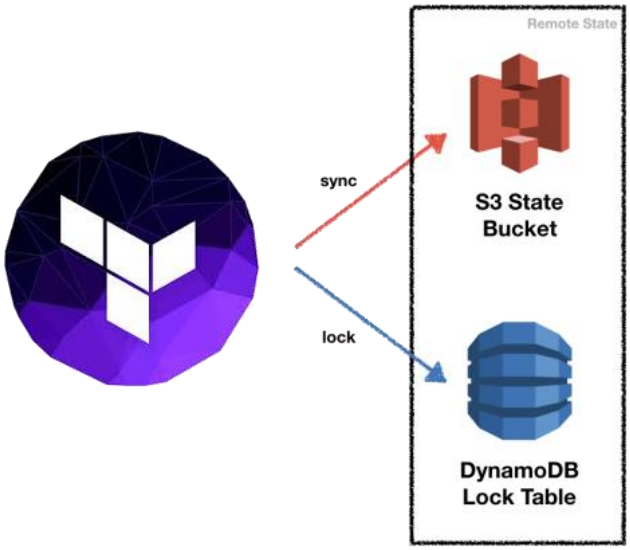
apply	Builds or changes infrastructure
console	Interactive console for Terraform interpolations
destroy	Destroy Terraform-managed infrastructure
env	Workspace management
fmt	Rewrites config files to canonical format
get	Download and install modules for the configuration
graph	Create a visual graph of Terraform resources
import	Import existing infrastructure into Terraform
init	Initialize a Terraform working directory
login	Obtain and save credentials for a remote host
logout	Remove locally-stored credentials for a remote host
output	Read an output from a state file
plan	Generate and show an execution plan
providers	Prints a tree of the providers used in the configuration
refresh	Update local state file against real resources
show	Inspect Terraform state or plan
taint	Manually mark a resource for recreation
untaint	Manually unmark a resource as tainted
validate	Validates the Terraform files
version	Prints the Terraform version
workspace	Workspace management

Module/data source and backend in terraform

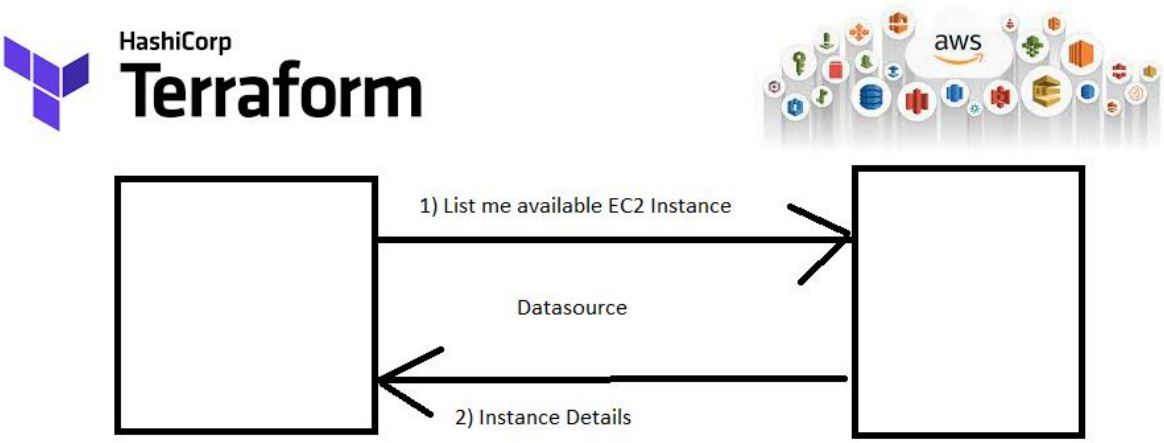
A *module* is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "3.4.0"
  # insert the 34 required variables here
}
```
Source Code: github.com/terraform-aws-modules/terraform-aws-ec2-instance

Terraform Backend: Backends define where Terraform's state snapshots are stored. A given Terraform configuration can either specify a backend, integrate with Terraform Cloud, or do neither and default to storing state locally.



Terraform datasource: In IT best practice is use existing system, instead of creating new system.



Terraform datasource is a querying mechanism from terraform to AWS, when they fetch existing system details.

Code:
```
data "aws_availability_zones" "az_available" {
  state = "available"}
```

Packer Json file:

```
{
    "variables":{
        "secret" :"",
        "access" : "",
        "region" : "us-west-2",
        "ami_name": "apache-from-packer"
    },
    "builders": [
        {
            "type": "amazon-ebs",
           "access_key": "{{user `access`}}",
           "secret_key": "{{user `secret`}}",
           "region": "{{user `region`}}",
            "source_ami": "ami-f2d3638a",
            "instance_type": "t2.micro",
            "ssh_username": "ec2-user",
            "ami_name": "{{user `ami_name`}}"
          }
    ],
    "provisioners":[
        {
            "type": "shell",
            "inline": [
                "sudo yum install httpd -y",
                "sudo service httpd start",
                "sudo chkconfig httpd on"]
          }
    ]
}
```

Terraform EC2 Creation file:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }
  }

  required_version = ">= 0.14.9"
}

provider "aws" {
  profile = "default"
  region  = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```