Automation Developer

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Lab page

https://jruels.github.io/automation-dev/

Terraform configuration

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform configuration

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

- Data sources
  - Queries AWS API for latest Ubuntu 16.04 image.
  - Stores results in a variable which is used later.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Resource: Defines specifications for creation of infrastructure.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- ami is set by results of previous data source query

Terraform configuration

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
variable "aws_instance_size" {
  default = "t2.small"
}
variable "aws_region" {
  default = "us-west-1"
}
```

- Define sane defaults in variables.tf

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Value is 'count' is setup by variable in variables.tf

Demo: Explore TF Configuration

Terraform CLI

The `terraform plan -refresh-only` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure but does modify the state file. If the state is changed, this may cause changes to occur during the next `plan` or `apply`.

Terraform CLI

Terraform is normally run from inside the directory containing the *.tf files for the root module. Terraform checks that directory and automatically executes them.

In some cases, it makes sense to run the Terraform commands from a different directory. This is true when wrapping Terraform with automation. To support that Terraform can use the global option `-chdir=...` which can be included before the name of the subcommand.

Terraform CLI

The `chdir` option instructs Terraform to change its working directory to the given directory before running the given subcommand. This means that any files that Terraform would normally read or write in the current working directory will be read or written in the given directory instead.

Terraform CLI

Command:

```
terraform -chdir=environments/dev (apply|plan|destroy)
```

Output:

```
❯ terraform apply
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the subcommand in the specified directory.

Terraform CLI

It can be time consuming to update a configuration file and run `terraform apply` repeatedly to troubleshoot expressions not working.

Terraform has a console subcommand that provides an interactive console for evaluating these expressions.

Terraform CLI

```
variable "x" {

default = [
        {
          name = "first",
          condition = {
            age = "1"
          }
          action = {
            type = "Delete"
          }
        },{
          name = "second",
          condition = {
            age = "2"
          }
          action = {
            type = "Delete"
          }
        }
    ]
}
```

Terraform CLI

Command:

```
terraform console
```

Output:

```
> var.x[1].name
"second"

var.x[0].condition
{
  "age" = "1"
}
```

Test expressions and interpolations interactively.

Terraform CLI

Terraform includes a graph command for generating visual representation of the configuration or execution plan. The output is in DOT format, which can be used by GraphViz to generate charts.

Terraform CLI

Command:

```
terraform graph
```

Output:

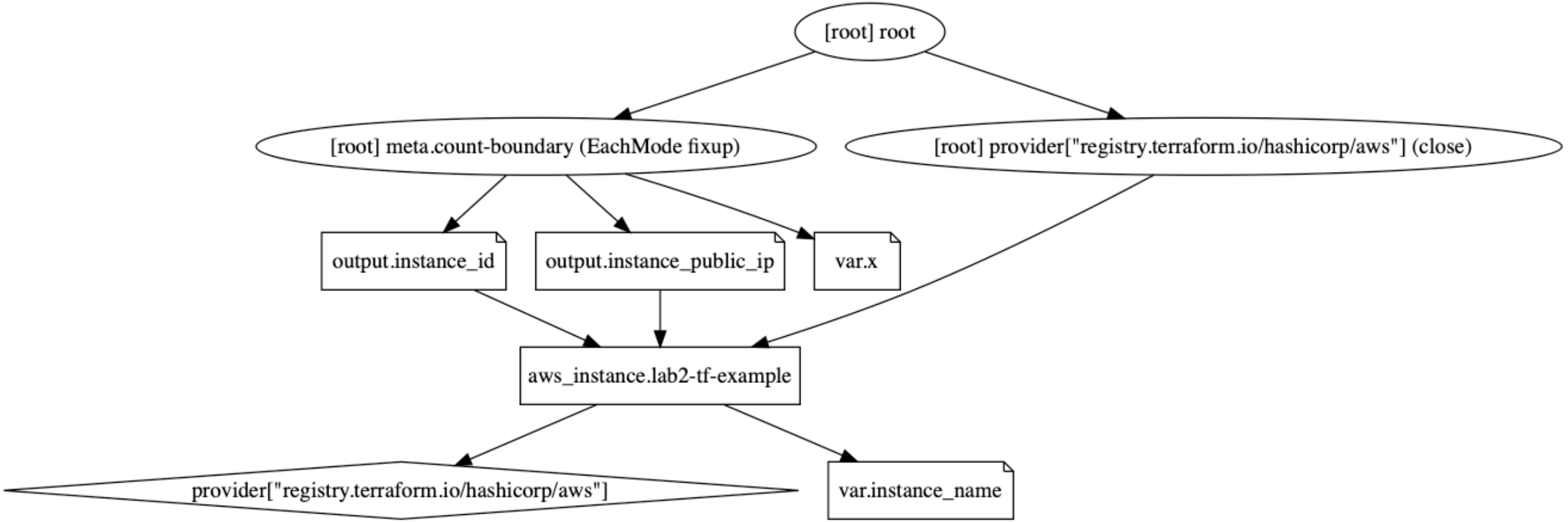
```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.lab2-tf-example (expand)"
[label = "aws_instance.lab2-tf-example", shape = "box"]
...
```

Create a visual graph of Terraform resources

```
terraform graph | dot -Tsvg > graph.svg
```

Terraform CLI

```
terraform graph| dot -Tsvg > graph.svg
```



HashiCorp Configuration Language (HCL)

ARM JSON:

```
"name" : "[concat(parameters('PilotServerName'), '3')]",
```

Terraform:

```
name = "${var.PilotServerName}3"
```

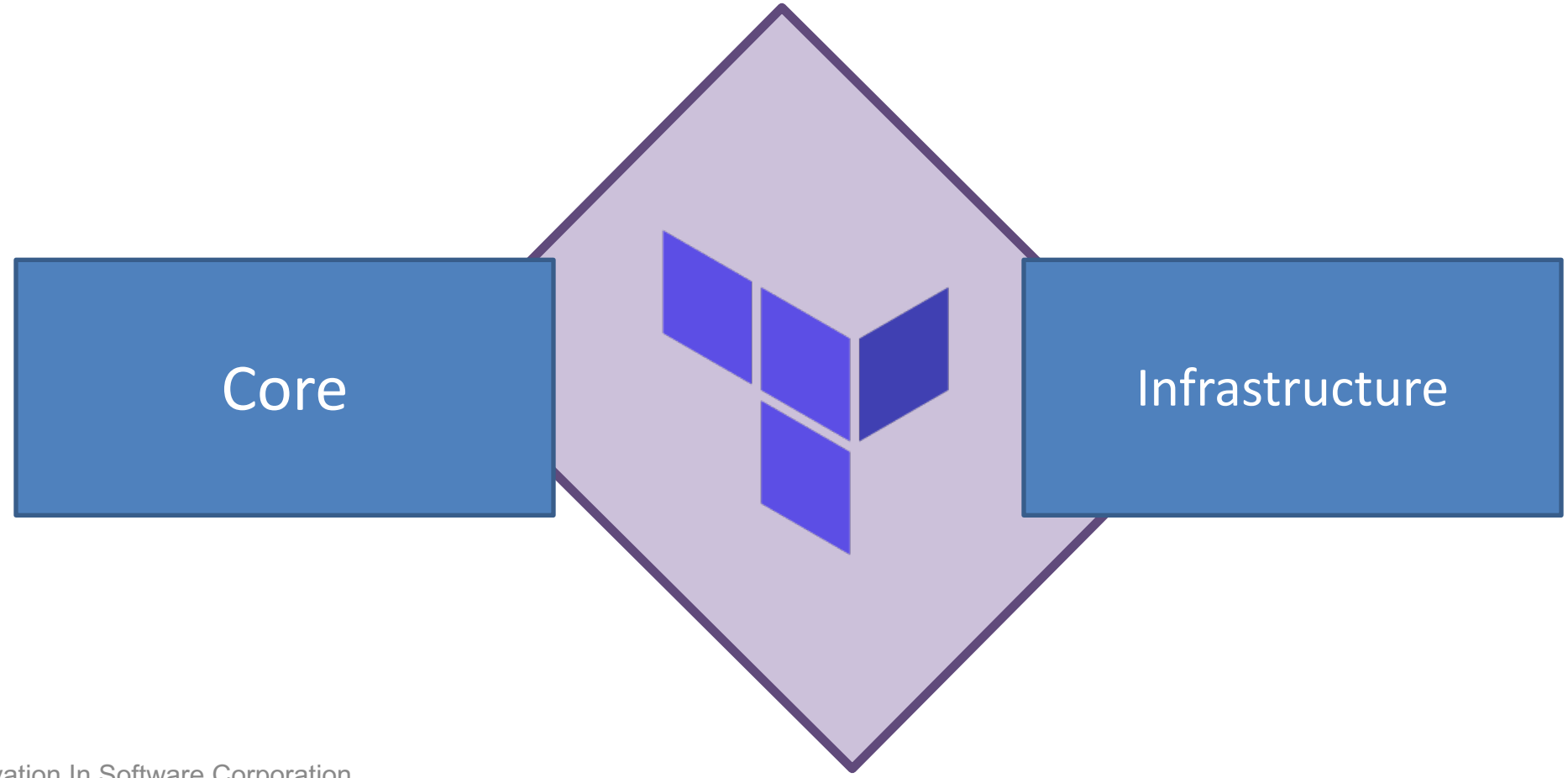
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers

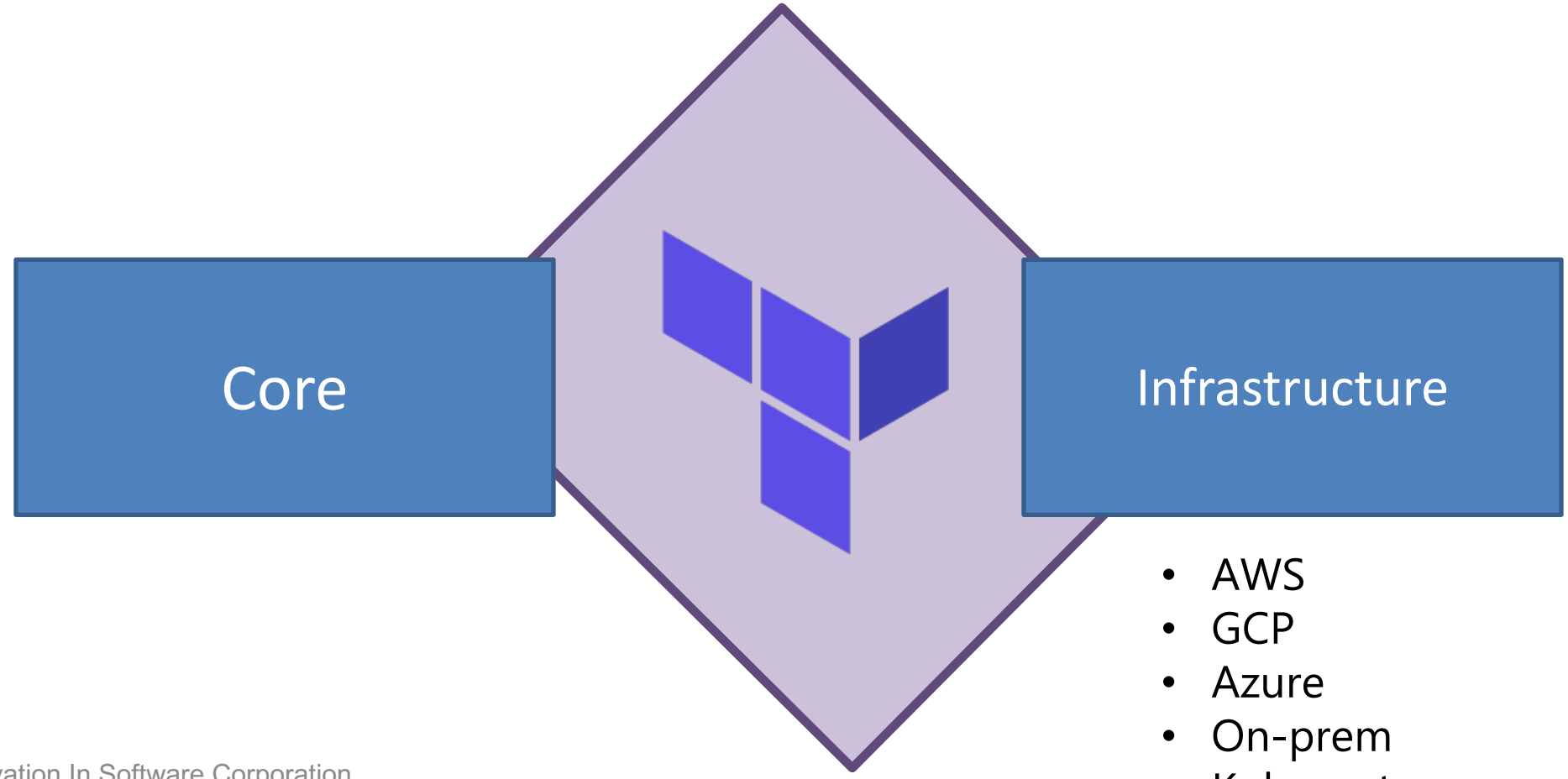
Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



- AWS
- GCP
- Azure
- On-prem
- Kubernetes

Directory hierarchy

Terraform has a pretty confusing file hierarchy. We are going to start with the basics.

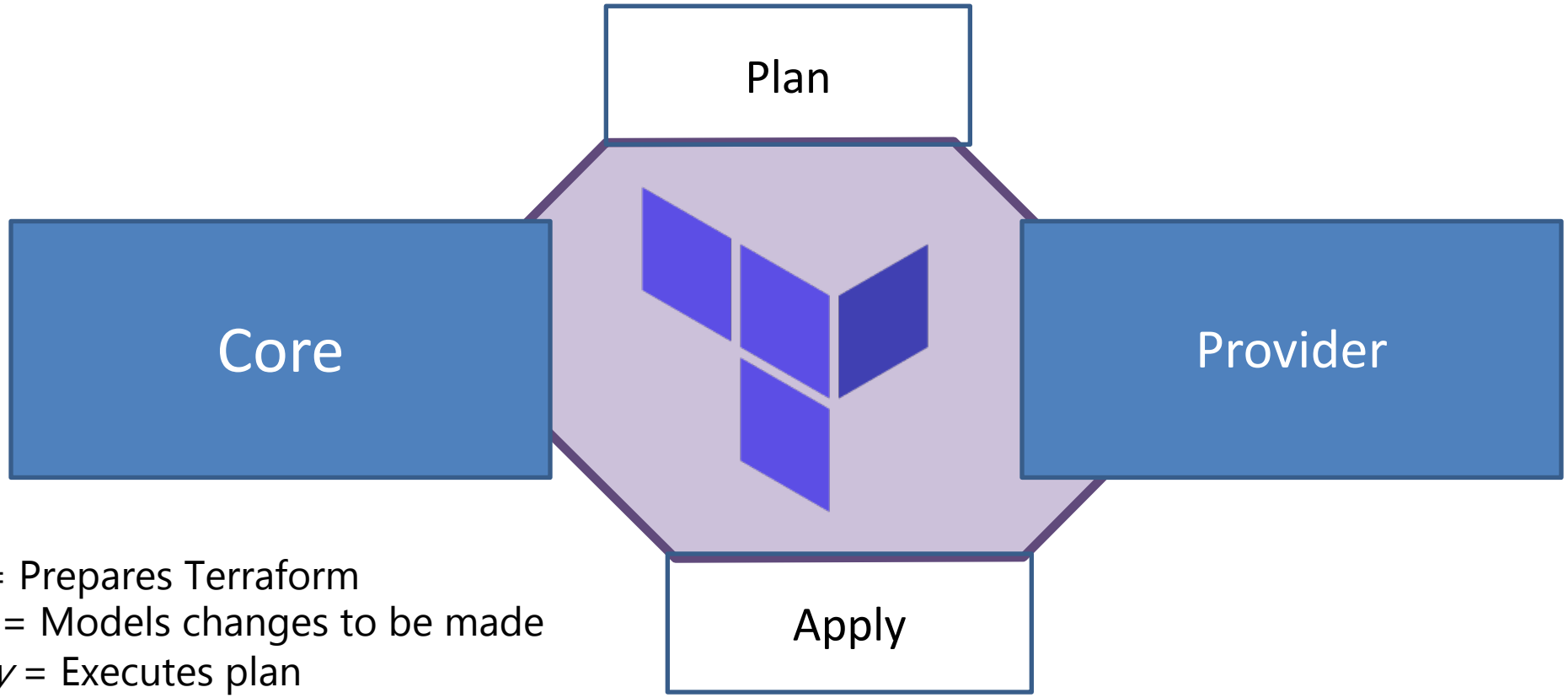
`provider.tf` - specify provider information

`main.tf` - create infrastructure resources.

`variables.tf` - define values for variables in main.tf

Terraform workflow

Terraform workflow consists of three common commands, Init, Plan and Apply.

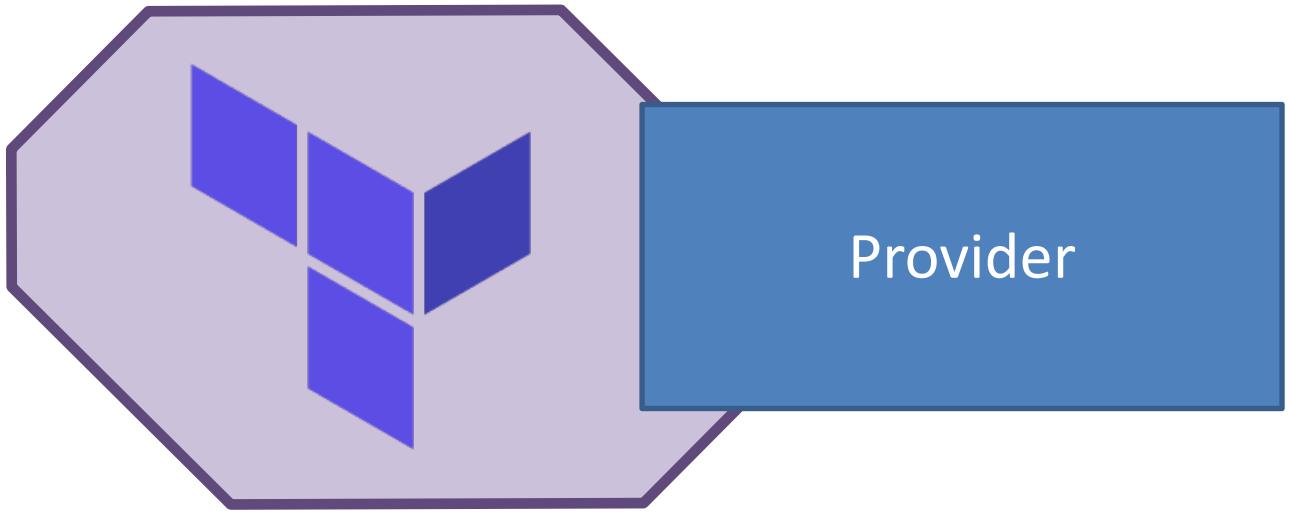


- *init* = Prepares Terraform
- *plan* = Models changes to be made
- *apply* = Executes plan

Terraform providers

Terraform has over 400 supported providers.

- Clouds (AWS, GCP, Rackspace, Azure etc.)
- Version control (GitHub, GitLab, Bitbucket)
- Software (Grafana, Consul, Docker, Kubernetes)
- more!



Terraform providers

Terraform relies on plugins called "providers" to interface with remote systems.

Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

Terraform providers

Each provider adds resources and/or data sources that Terraform manages.

Every resource type is implemented by a provider. Terraform can't manage any kind of infrastructure without providers.

Providers are used to configure infrastructure platforms (either cloud or self-hosted). Providers also offer utilities for tasks like generating random numbers for unique resource names.

Terraform provider configuration

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

Terraform provider configuration

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Terraform supports 'aliases' for multiple configurations for the same provider, and you can select which one to use on a per-resource or per-module basis.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
}
```

Terraform allows you to configure the providers in code. configure specific versions or pull in latest.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
  subscription_id = "SUBSCRIPTION-ID"
  client_id       = "CLIENT-ID"
  client_secret   = "CLIENT-SECRET"
  tenant_id       = "TENANT_ID"
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Terraform provider configuration

For Azure that can be az login, Managed Service Identity, or environment variables.

```
az login
```

```
export ARM_TENANT_ID=
export ARM_SUBSCRIPTION_ID=
export ARM_CLIENT_ID=
export ARM_CLIENT_SECRET=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {}
```

```
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  region                   = "us-west-2"
  shared_credentials_file  = "~/.aws/creds"
  profile                  = "customprofile"
}
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  access_key     = "MYKEY"
  secret_key     = "MYSECRET"
}
```

Terraform ASA provider

```
provider "ciscoasa" {
  api_url           = "https://10.0.0.5"
  username          = "admin"
  password.         = "YOUR SECRET PASSWORD"
  ssl_no_verify     = false
}
```

Providers allow you to authenticate in the code block.
This is a very BAD idea.
Use environment variables:
CISCOASA_USER
CISCOASA_PASSWORD

Terraform ASA ACL resource

```
resource "ciscoasa_acl" "foo" {
  name = "aclname"
  rule {
    source                = "192.168.10.5/32"
    destination           = "192.168.15.0/25"
    destination_service = "tcp/443"
  }
  rule {
    source                = "192.168.10.0/24"
```

Example of creating ACL

Terraform ASA static route resource

```
resource "ciscoasa_static_route" "ipv4_static_route" {
  interface               = "inside"
  network                 = "10.254.0.0/16"
  gateway                 = "192.168.10.20"
  }

resource "ciscoasa_static_route" "ipv6_static_route" {
  interface               = "inside"
  network                 = "fd01:1337::/64"
  gateway                 = "fd01:1338::1"
```

Terraform Resources

Terraform resources

Resources are the most important element of the Terraform language. Each resource block describes one or more infrastructure objects, such as networks, instances, or higher-level objects such as DNS records.

Terraform resources

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web"). The name is used to refer to this resource from elsewhere in the same Terraform module but has no significance outside that module's scope.

The resource type and name together serve as an identifier for a given resource and so must be unique within a module.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

resource = top level keyword

Terraform resources

Within the block body (between { and }) are the configuration arguments for the resource itself. Most arguments in this section depend on the resource type, and indeed in this example both ami and instance_type are arguments defined specifically for the aws_instance resource type.

Terraform resources

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Configuration data for the "aws_instance" resource:

- AMI
- Instance Type (flavor)

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

type = this is the name of the resource. The first part tells you which provider is belongs to. Example: azurerm_virtual_machine. This means the provider is Azure and the specific type of resources is a virtual machine.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

name = arbitrary name to refer to resource. Used internally by TF and cannot be a variable.

Resources (building blocks)

Create an Azure Resource Group.

- Azure requires all resources be assigned to a resource group.

```
resource "azurerm_resource_group" "training" {
  name     = "training-workshop"
  location = "westus"
}
```

Resources (building blocks)

TIP: You can assign random names to resources.

```
resource "random_id" "project_name" {
  byte_length = 4
}
resource "azurerm_resource_group" "training" {
  name       = "${random_id.project_name.hex}-training"
  location   = "westus"
}
```

Lab: Create an instance

Terraform resource customization

Some resource types provide a special timeouts nested block argument that allows you to customize how long certain operations are allowed to take before being considered to have failed. For example, aws_db_instance allows configurable timeouts for create, update and delete operations.

Terraform resources

Timeouts are handled entirely by the resource type implementation in the provider, but resource types offering these features follow the convention of defining a child block called timeouts that has a nested argument named after each operation that has a configurable timeout value. Each of these arguments takes a string representation of a duration, such as "60m" for 60 minutes, "10s" for ten seconds, or "2h" for two hours.

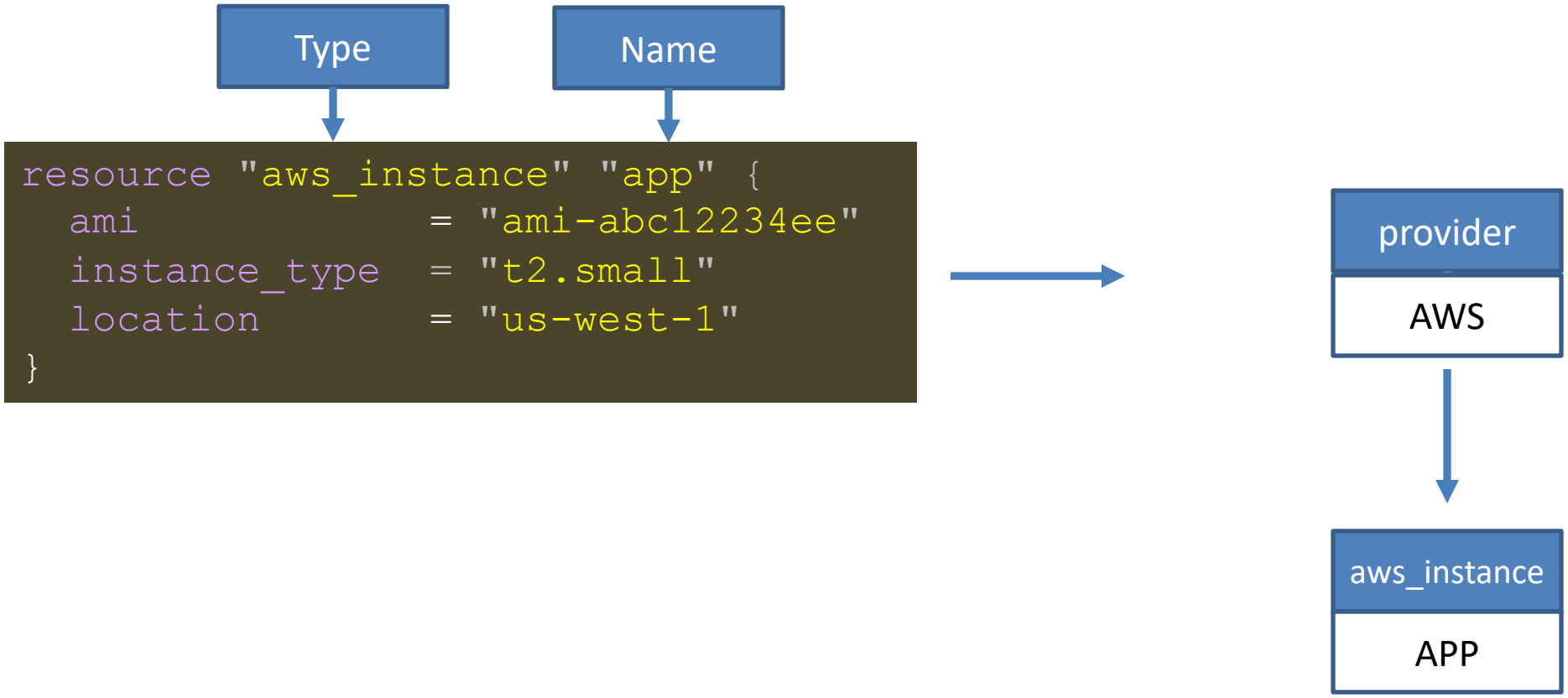
Terraform resources

```
resource "aws_db_instance" "example" {
  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

Timeout arguments

Resources



Terraform Variables

Terraform variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

Terraform variables

Each input variable accepted by a module must be declared using a variable block.

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform variables

Terraform has some reserved variables:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- locals

- These are reserved for meta-arguments in module blocks.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform variables

Optional variable arguments:

- default: A default value which makes the variable optional
- type: Specifies value type accepted for the variable.
- description: Specifies the variable's documentation
- validation: A block to define validation rules
- sensitive: Does not show value in output

Terraform variables

Set default values for variables in variables.tf
If default values are omitted, user will be prompted.

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
```

Terraform variables (type constraint)

Type constraints are optional but highly encouraged. They can serve as reminders for users of the module and help Terraform return helpful errors if the wrong type is used.

The type argument allows you to restrict acceptable value types. If no type constraint is defined, then a value of any type is accepted.

The keyword any may be used to indicate that any type is acceptable.

If both the type and default arguments are specified, the given default value must be convertible to the specified type.

Terraform variables (type constraint)

Type constraints are created from a mixture of type keywords and constructors.

Supported type keywords:
- string
- number
- bool

- Type constructors allow you to specify complex types such as collections:
- list (<TYPE>)
- set (<TYPE>
- map (<TYPE>)
- object ({<ATTR NAME> = <TYPE>, ...})
- tuple ([<TYPE>, ...])

Terraform variables

Input variables are part of its user interface. You can briefly describe the purpose of each variable using the optional description argument.

```
## Variable description example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"
}
```

Terraform variables (validation)

In addition to Type Constraints, you can specify arbitrary custom validation rules for a variable.

```
## Variable validation example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"


  validation {
    condition = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
     error_message = "The image_id value must be a valid AMI id, starting with
\"ami-\"."
  }
}
```

Terraform variables (validation)

A condition argument is an expression that must use the value of the variable to return true if the value is valid, or false if it is invalid.

The expression can refer only to the variable that the condition applies to and must not produce errors.

Terraform variables (validation)

If the failure of an expression is the basis of the validation decision, use the can function to detect such errors. For example:

```
## Variable validation example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"
}

  validation {
    # regex(...) fails if it cannot find a match
    condition = can(regex("^ami-", var.image_id))
    error_message = "The image_id value must be valid."
```

Terraform variables

The `variables.tf` file includes default values for variables.

```
variable "region" {
  description = "Region to deploy Jenkins into"
  default     = "us-west2"
}

variable "jenkins_initial_password" {
  description = "Jenkins user password"
  default     = "bitnami"
}

variable "project_id" {
  description = "Project for Jenkins VM"
}
```

Terraform output values

Output values are like the return values of a Terraform module, and have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.

Lab: Variables and output

Lab: Create multi-resource infrastructure

Terraform state

Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored, etc.

Terraform state

When starting out with Terraform the best approach is to stick with a local backend. It also makes sense to use a local backend if you are the only one managing the infrastructure.

Terraform state

Backend configuration is only used by Terraform CLI.

Terraform Cloud and Terraform Enterprise always use their own state storage when performing Terraform runs, so they ignore any backend block in the configuration.

It's common to use Terraform CLI with Terraform Cloud, so best practice is to include a backend block in the configuration with a remote backend pointing to the relevant Terraform Cloud workspace(s)

Terraform state

Terraform supports local and remote state file storage.

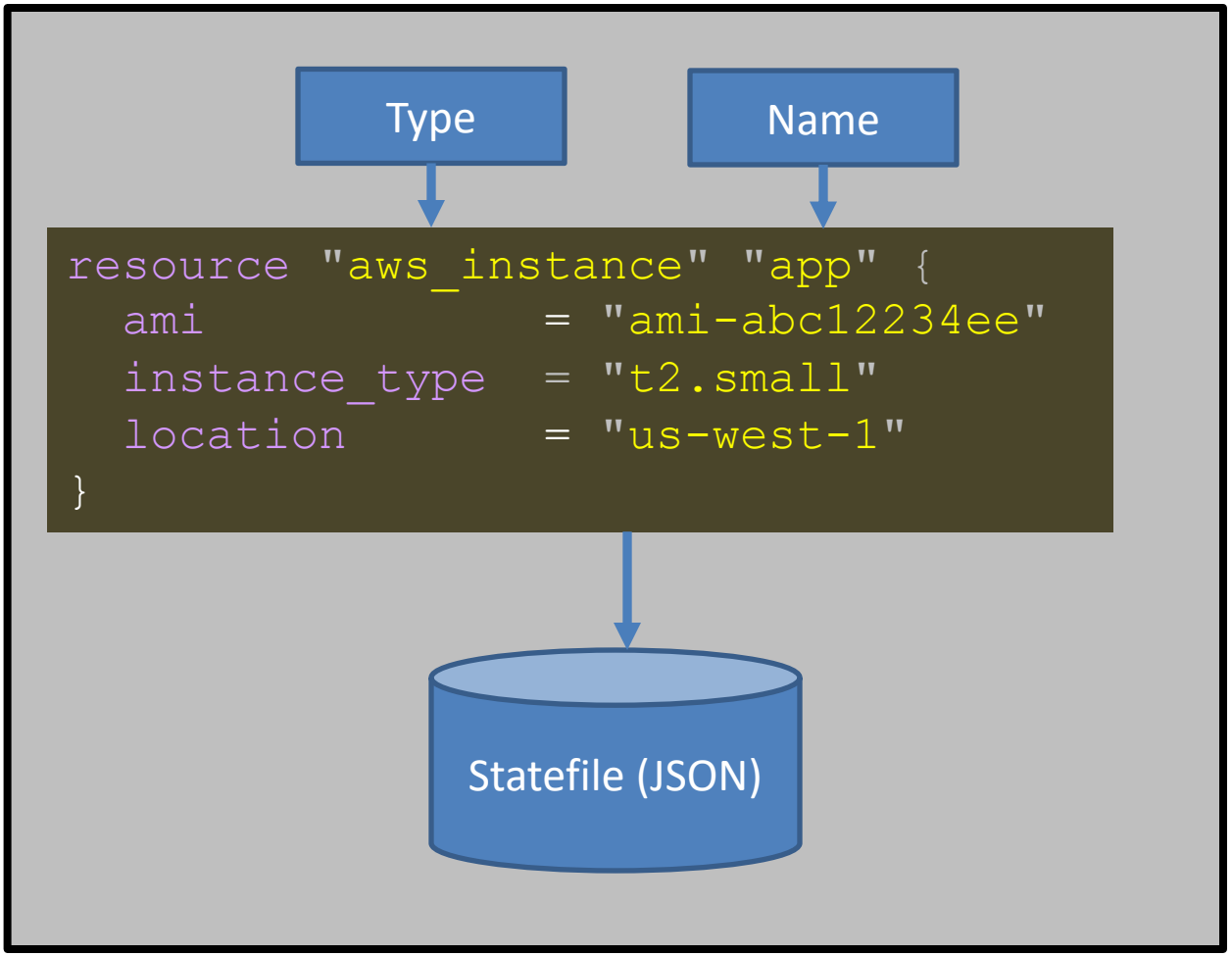
- Default is local storage
- Remote backends:
  - S3, Azure Storage, Google Cloud Storage

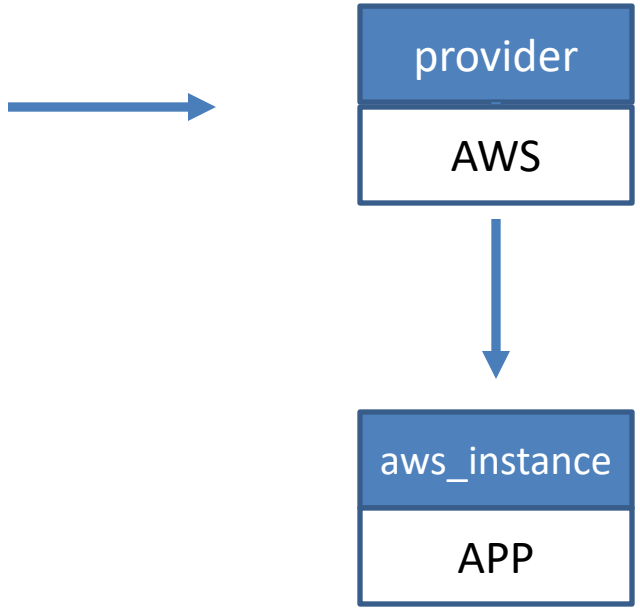
Terraform state

Terraform supports many backends:

- local (default)
- remote (Terraform Enterprise/Terraform Cloud)
- azurerm
- consul
- s3
- gcs
- more...

Terraform state (local)



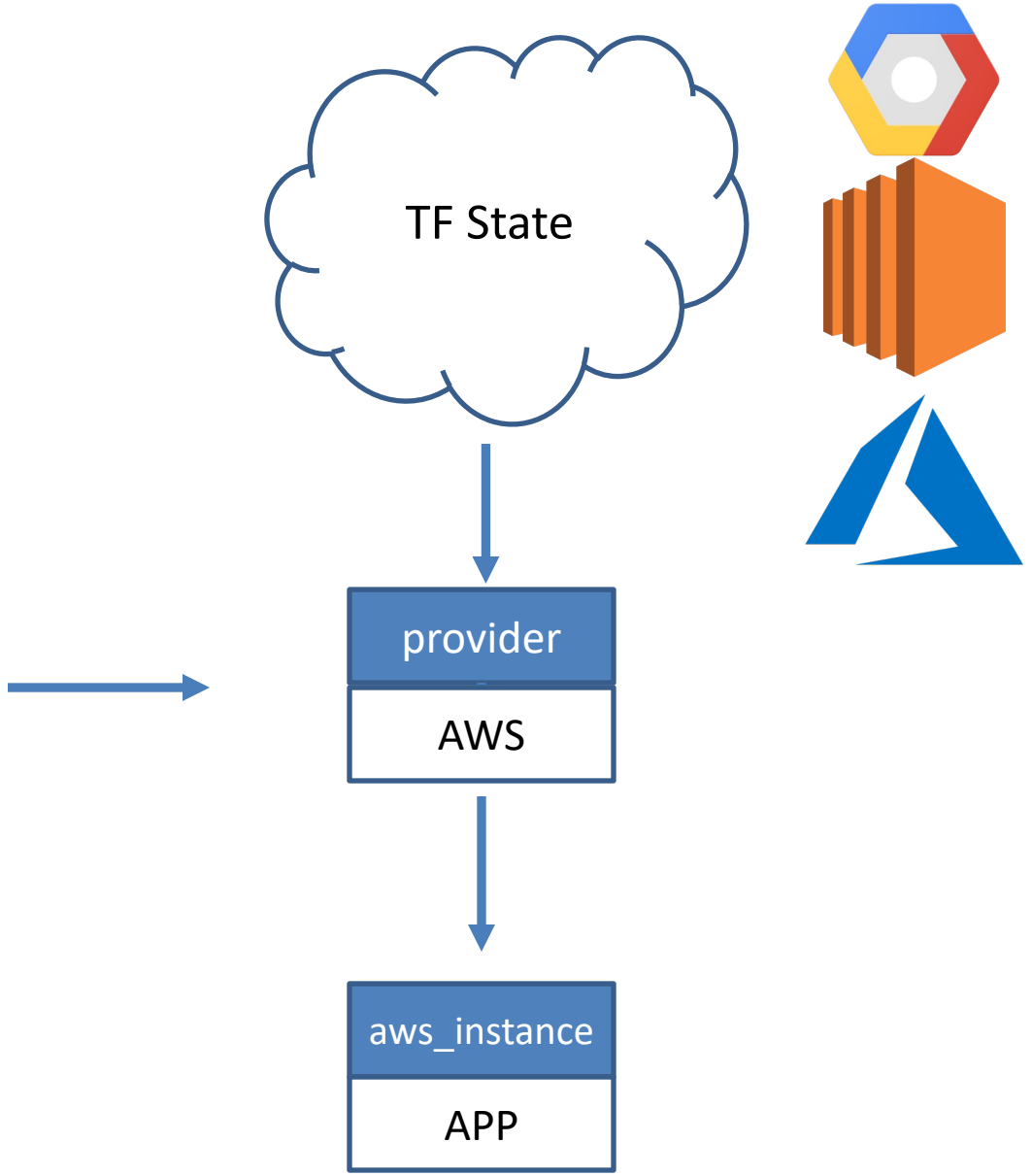


Terraform state (cloud)

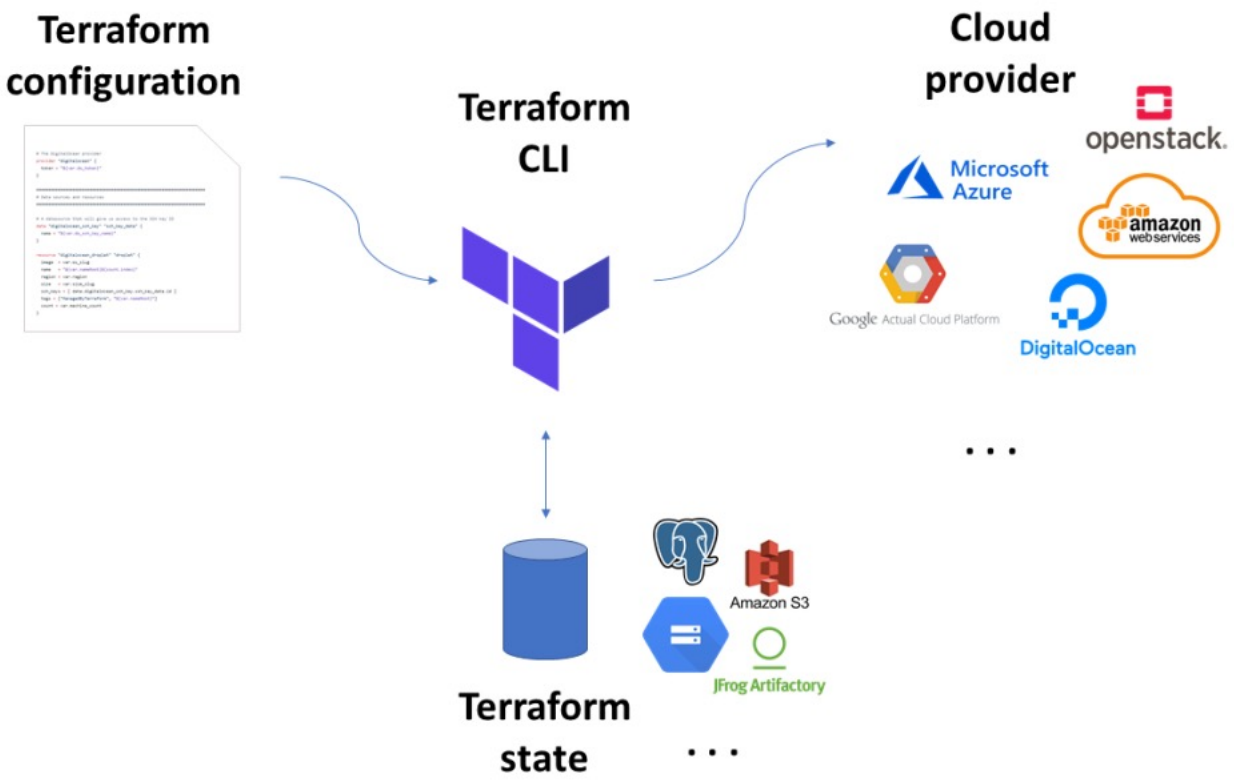
Type

Name

```
resource "aws_instance" "app" {
  ami             = "ami-abc12234ee"
  instance_type   = "t2.small"
  location        = "us-west-1"
}
```



Terraform state



Terraform state locking

Terraform will lock your state for all operations that could write state if your backend supports it. This prevents others from acquiring the lock and possibly corrupting your state.

State locking automatically happens on all operations that write state.

- azurerm
- consul
- etcdv3
- gcs
- s3 (locking via DynamoDB)

Terraform state backend

Backends are configured with a nested "backend" block within the top-level terraform block.

```
terraform {
  backend "remote" {
    organization              = "foo"
}
```

Terraform Modules

Don't Repeat Yourself (DRY)

DRY is a principle that discourages repetition, and encourages modularization, abstraction, and code reuse. Applying it to Terraform, using modules is a big step in the right direction.

However, repetitions still happen. You may end up having virtually the same code in different environments, and when you need to make one change, you have to make that change many times.

Don't Repeat Yourself (DRY)

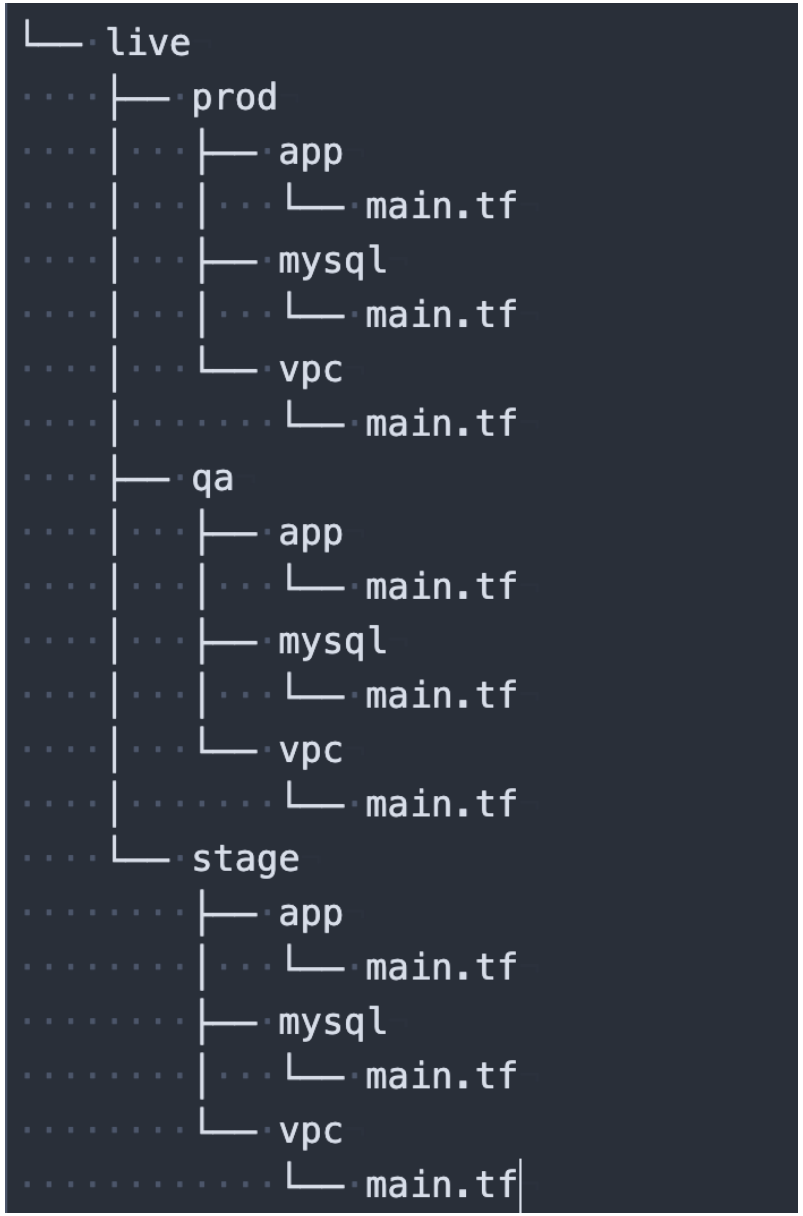
This problem can be addressed in a few ways. A recommended approach is to create a folder for shared or common files and then create symlinks to these files from each environment. This way, you can make a change to the common file(s) once and it is applied in all the environments.

Don't Repeat Yourself (DRY)

There is also an open-source tool, Terragrunt, which solves the same problem differently. It is a wrapper around the Terraform CLI commands, which allows you to write your Terraform once and then, in a separate repository, define only input variables for each environment - no need to repeat Terraform code for each environment. Terragrunt is also handy for orchestrating Terraform in CICD pipelines for multiple separate projects.

Don't Repeat Yourself (DRY)

Common directory structure for managing three environments (prod, qa, stage) with the same infrastructure (an app, a MySQL database and a VPC)



Don't Repeat Yourself (DRY)

The contents of each environment will be almost identical, except for perhaps a few settings (e.g. the prod environment may run bigger or more servers). As the size of the infrastructure grows, having to maintain all this duplicated code becomes more error prone.

Modules

Modules are Terraforms way of managing multiple resources that are used together. A module consists of a collection of `.tf` files kept together in a directory.

Modules

As you use Terraform to manage your infrastructure, you will create increasingly complex configurations.

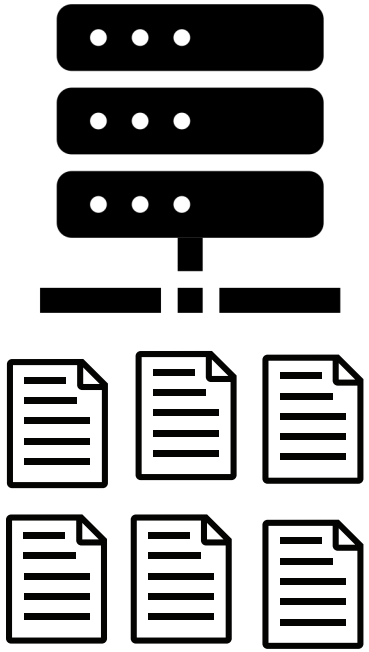
There is no limit to the complexity or size of a single Terraform configuration file or directory, so it's possible to include everything in one directory or even one file.

Modules

If you take a "monolithic" approach, and include all configuration in one file or directory you may run into these problems:

- Understanding and navigating the configuration becomes difficult.
- Making an update to one section may cause unintended consequences in other sections.
- There will be increasing duplication of similar blocks of configuration!
    - Dev, Staging, Production

Modules

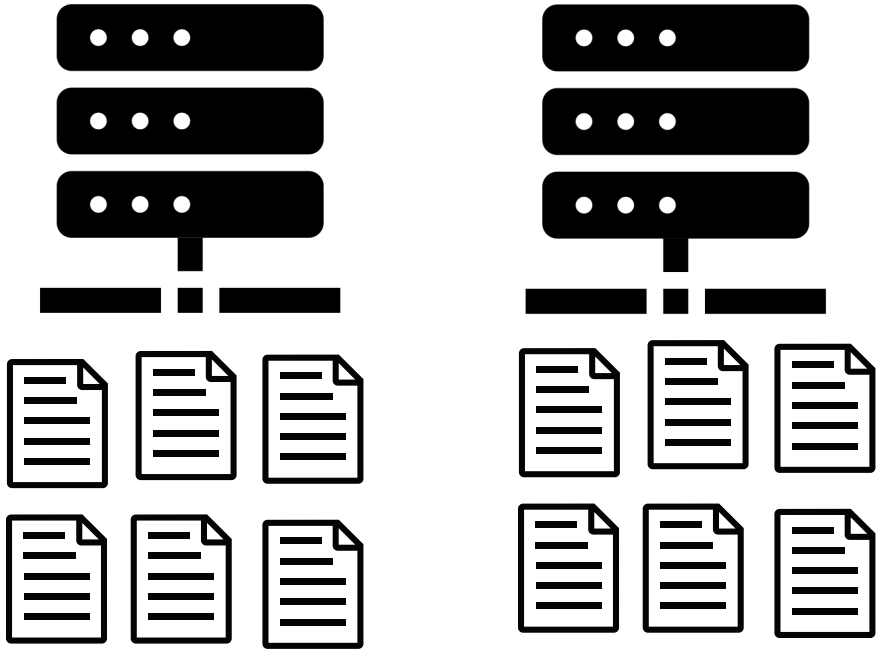


Dev

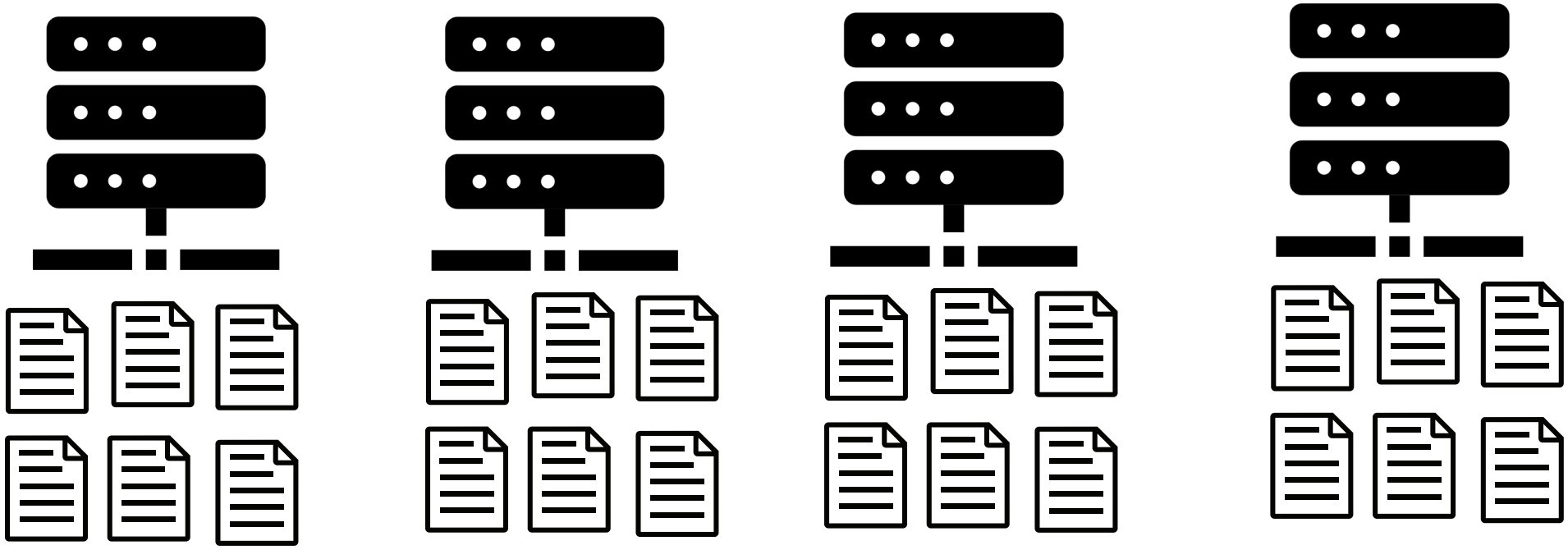
One environment can have many configuration files:

- main.tf
- variables.tf
- backend.tf
- provider.tf
- myvariables.tfvar
- outputs.tf

Modules



Modules



Modules

How modules help:

- Organize configuration - Modules make it easier to navigate, understand and update your configuration by keeping related parts of the configuration together.

- Group the configuration into logical components.

Modules

- Encapsulate configuration - Modules encapsulate the configuration into distinct logical components. This prevents unintended consequences, such as a change in one part affecting other parts of the infrastructure.
- Reduces simple errors like using the same name for two different resources.

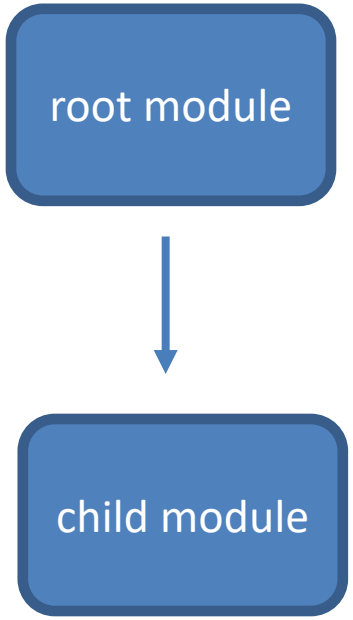
Modules

- Re-use configuration - Writing all configuration from scratch is time consuming and error prone. Modules save time and reduce errors by re-using configuration written by you, other members of your team, or experts who published modules in the registry.
- Save time!
  - Don't reinvent the wheel

Modules

- Consistency - Modules help provide consistency in your configurations. Consistency is important for readability of configuration, but also helps to ensure best practices are applied to all the configuration.
- Writing configuration from scratch can lead to security issues by having misconfigured attributes:
  - S3, Google Cloud Buckets
  - Security groups, old AMIs

Modules



- Provided by you:
- main.tf
- variables.tf
- outputs.tf

- Provided by module:
- resources
  - aws_instance
  - elb
  - ebs
  - ...

Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory. However, your configuration can use module blocks to call modules in other directories. When Terraform encounters a module block, it loads and processes that module's configuration files.

Modules

- Modules can be loaded from a local directory, or from a remote source.
  - GitHub, Private registry, Public registry
  - The Terraform public registry includes:
    - Official modules: maintained by Terraform employees
    - Verified: maintained by vendor employees
    - Community: maintained by community volunteers.

Modules

- Best practices
  - Start writing with modules in mind. Even for modestly complex configurations managed by a single person, you'll find the many benefits of using modules outweigh the time it takes to use them properly.
  - Use the public Terraform Registry to find useful modules (if company policy allows it).
    - No need to reinvent the wheel
    - Reduce burden of writing/maintaining configuration from scratch.

Modules

- Best practices
  - Publish and share modules with your team! Infrastructure is managed by a team of people, and modules are an important way teams can work together to create and maintain infrastructure.

Lab: Write your own module

Lab: Refactor monolithic codebase