Infrastructure as Code (IaC)

Terraform

Dr. Assane Wade
Abir Chebbi
2023-2024

hepia
Haute école du paysage, d'ingénierie et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée de Suisse occidentale

Infrastructure as Code (IaC)

- The managing and provisioning of infrastructure through code instead of through manual processes.
- Uses a high-level descriptive coding language to automate the provisioning of IT infrastructure.

Infrastructure as Code benefits

- Faster time to production/market
- Improved consistency—less ‘configuration drift’
- Faster, more efficient development
- Protection against churn
- Lower costs and improved ROI

Declarative vs. imperative approach

Declarative approach (or Functional approach) is the optimal method:

> You articulate the ultimate state you want for the infrastructure you are provisioning, and the Infrastructure as Code (IaC) software takes care of the remaining tasks. This includes tasks such as initiating the virtual machine (VM) or container, as well as installing and configuring required software, resolving interdependencies between system and software components, and overseeing versioning.

```
resource "openstack_compute_instance_v2"
"app_server" {
    name             = "TF-managed"
    image_id         = "cirros"
    flavor_name      = "m1.tiny"
    }
```

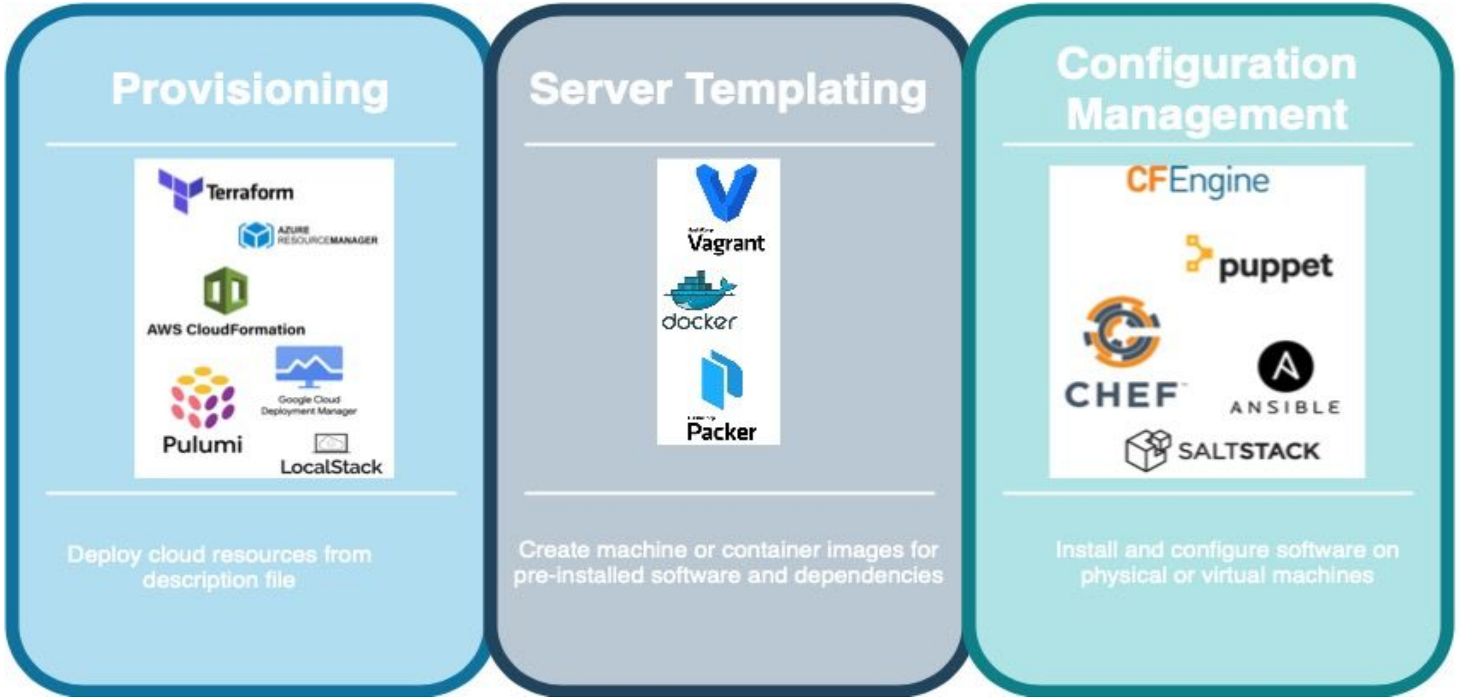
Declarative vs. imperative approach

Imperative approach (procedural approach):

The solution facilitates the creation of automation scripts designed to provision your infrastructure incrementally, addressing each specific step individually. Although this approach may involve more management effort as you scale, it offers the advantage of being more comprehensible for existing administrative staff. Additionally, it allows for the utilization of configuration scripts that are already in place, streamlining the integration process.

```
image = nova_client.images.find(name="cirros")
flavor = nova_client.flavors.find(name="m1.tiny")
instance = nova_client.servers.create(name="vm2",
image=image, flavor=flavor,...)
```

Infrastructure as Code (IaC) Tools


Provisioning
Terraform
AZURE RESOURCE MANAGER
AWS CloudFormation
Pulumi
Google Cloud Deployment Manager
LocalStack
Deploy cloud resources from description file
Server Templating
Vagrant
docker
Packer
Create machine or container images for pre-installed software and dependencies
Configuration Management
CFEngine
puppet
CHEF
ANSIBLE
SALTSTACK
Install and configure software on physical or virtual machines

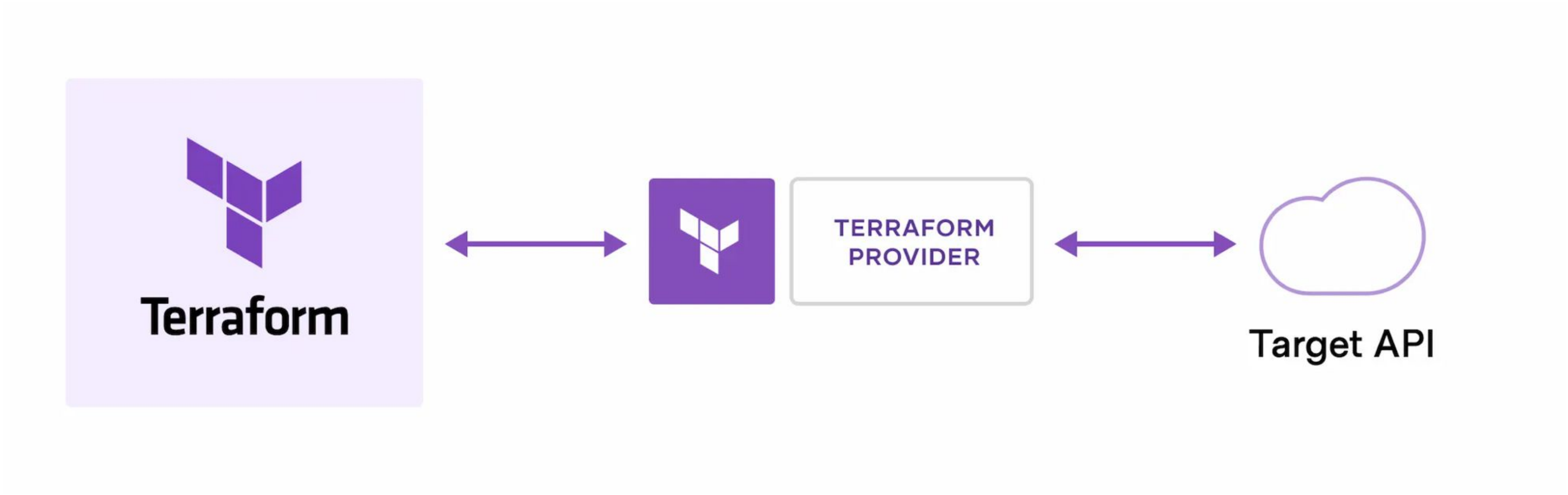
Terraform

- HashiCorp Terraform is a tool for infrastructure as code, allowing users to define and provision resources using human-readable configuration files.
- It works with both cloud and on-premises resources, handling low-level components like compute and storage, as well as higher-level features such as DNS entries and SaaS functionalities.

Terraform Characteristics

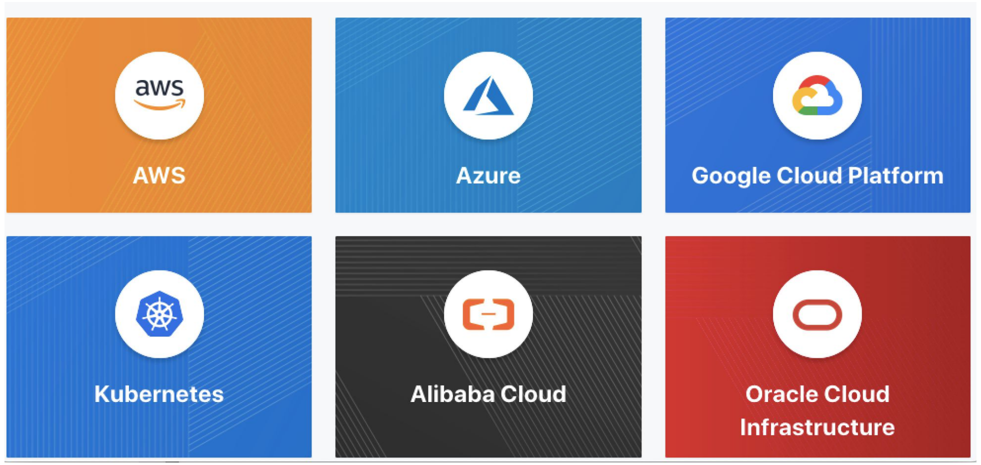
- Declarative configuration language
- Provider-agnostic support for various cloud providers
- Resource graph for dependency management
- Execution plans for change preview
- State management
- Modules for code reuse
- Integration capabilities with CI/CD pipelines

Terraform Basic Components



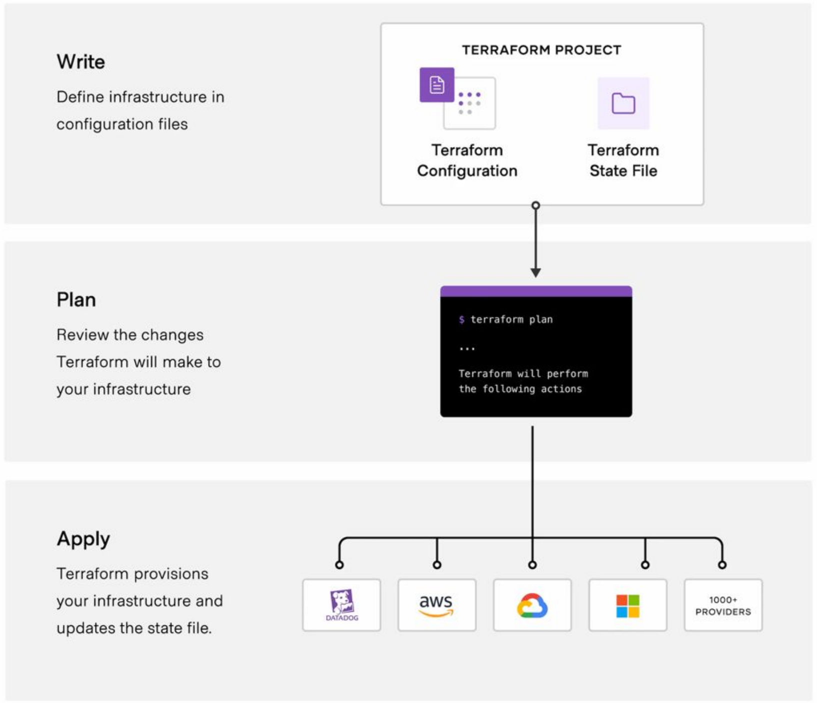
Terraform Providers

- Terraform Registry
- Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, and many more.



Terraform Stages

- **Write**: You define resources, which may be across multiple cloud providers and services.
- **Plan**: Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
- **Apply**: On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies.


Write
Define infrastructure in configuration files
TERRAFORM PROJECT
Terraform Configuration
Terraform State File
Plan
Review the changes Terraform will make to your infrastructure
Apply
Terraform provisions your infrastructure and updates the state file.
DATADOG
aws
1000+ PROVIDERS

```
$ terraform plan
...
Terraform will perform
the following actions
```

Terraform’s Language

- Terraform's language is its primary user interface.

- Configuration files you write in Terraform language tell Terraform what plugins to install, what infrastructure to create, and what data to fetch.

- Terraform language also lets you define dependencies between resources and create multiple similar resources from a single configuration block.

Terraform Files and Directories

- Code in the Terraform language is stored in plain text files with the ***.tf*** file extension.
- JSON-based variant of the language with the ***.tf.json*** file extension.
- Files containing Terraform code are called configuration files
- Configuration files must always use UTF-8 encoding

Directories and Modules

- A module is a collection of .tf and/or .tf.json files kept together in a directory.
- Nested directories are treated as completely separate modules

Override Files

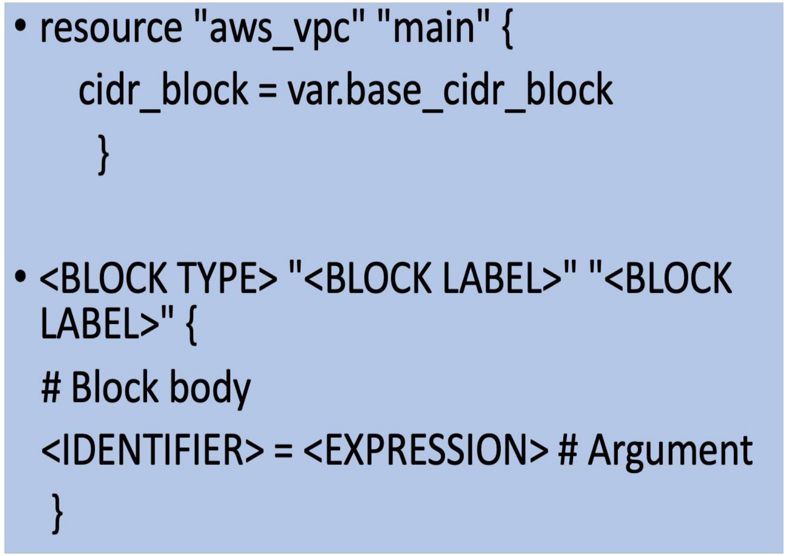
- Terraform normally loads all of the ***.tf*** and ***.tf.json*** files within a directory and expects each one to define a distinct set of configuration objects. If two files attempt to define the same object, Terraform returns an error.
- Exception: Terraform has special handling of any configuration file whose name ends in _***override.tf*** or _***override.tf.json***. This special handling also applies to a file named literally override.tf or override.tf.json.

Terraform Syntax

- **Configuration Syntax** : describes the native grammar of the Terraform language.
- **JSON Configuration Syntax** : documents how to represent Terraform language constructs in the pure JSON variant of the Terraform language.
- **Style Conventions** : documents some commonly accepted formatting guidelines for Terraform code. These conventions can be enforced automatically with terraform fmt.

Terraform Syntax

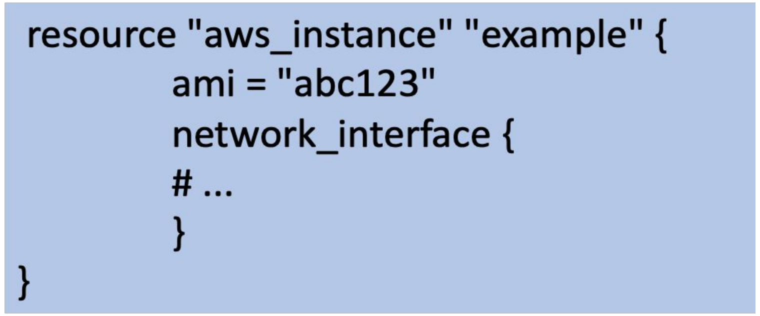
- Blocks are containers for other content and usually represent the configuration of some kind of object, like a resource.
- Arguments assign a value to a name. They appear within blocks.
- Expressions represent a value, either literally or by referencing and combining other values. They appear as values for arguments, or within other expressions.



```
• resource "aws_vpc" "main" {
     cidr_block = var.base_cidr_block
    }

• <BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK
  LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
   }
```

Blocks

- A block is a container for other content



```
resource "aws_instance" "example" {
          ami = "abc123"
          network_interface {
          # ...
          }
}
```

- A block has a type (resource in this example). Each block type defines how many labels must follow the type keyword. The resource block type expects two labels, which are aws_instance and example in the example above.
- A particular block type may have any number of required labels, or it may require none as with the nested network_interface block type.

Resource Syntax

- A "resource" block declares a resource of a specific type with a specific local name. The name is used to refer to this resource in the same Terraform module but has no meaning outside that module's scope.
- The resource type ("aws_instance") and name ("example") together must be unique within a module because they serve as an identifier for a given resource.
- The arguments often depend on the resource type

Eg: "aws_instance” has arguments including: ami, instance_type

Arguments

- An argument assigns a value to a particular name

  Eg: image_id = "abc1w3”

- The identifier before the equals sign is the argument name, and the expression after the equals sign is the argument's value.

Comments

- # begins a single-line comment, ending at the end of the line.
- // also begins a single-line comment, as an alternative to #.
- /* and */ are start and end delimiters for a comment that might span over multiple lines.

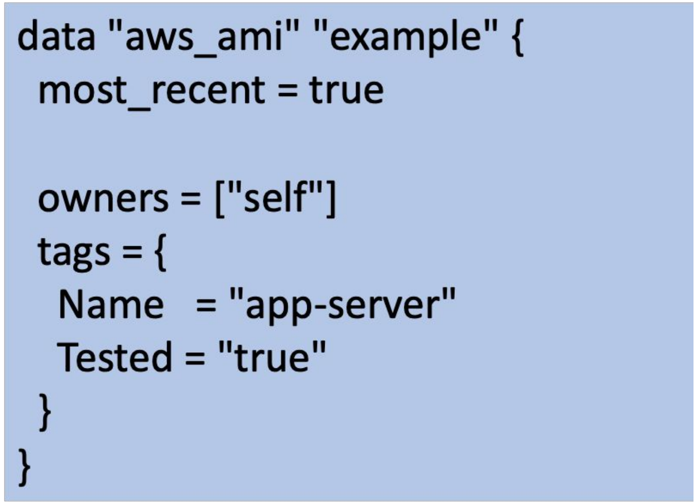
Meta-Arguments

The Terraform language defines the following meta-arguments, which can be used with any resource type to change the behavior of resources:

- depends_on
- count
- for_each
- lifecycle
- provisioner

Data Sources

- Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.
- Each provider may offer data sources alongside its set of resource types.



```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

Variables and Outputs

- Input Variables serve as parameters for a Terraform module, so users can customize behavior without editing the source.
- Output Values are like return values for a Terraform module.
- Local Values are a convenience feature for assigning a short name to an expression.

Input variables

- Input variables let you customize aspects of Terraform modules without altering the module's own source code.
- This functionality allows you to share modules across different Terraform configurations, making your module composable and reusable.
- 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.
- Input variables are like function arguments.

Variable declaration

- The Name 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.
- Name of a variable can be any valid identifier except the following: source, version, providers, count, for_each, lifecycle, depends_on, locals.

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

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

Variables Arguments

- Default : A default value which then makes the variable optional.
- Type: This argument specifies what value types are accepted for the variable.
- Description: This specifies the input variable's documentation.
- Validation: A block to define validation rules, usually in addition to type constraints.
- Sensitive: Limits Terraform UI output when the variable is used in configuration.
- Nullable: Specify if the variable can be null within the module.

Variables Arguments - Type Constraints

Type keywords:

- String
- Number
- Bool

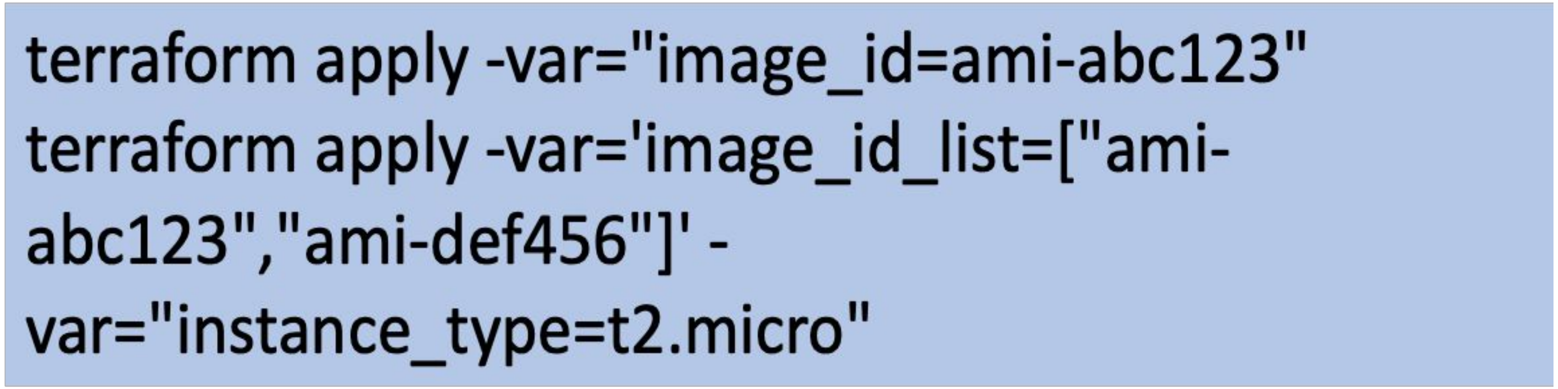
The type constructors allow you to specify complex types such as collections:

- list(<TYPE>)
- set(<TYPE>)
- map(<TYPE>)
- object({<ATTR NAME> = <TYPE>, ... })
- tuple([<TYPE>, ...])

Assigning Values to Root Module Variables

When variables are declared in the root module of your configuration, they can be set in a number of ways:

- In a Terraform Cloud workspace.
- Individually, with the -var command line option.



```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-
abc123","ami-def456"]' -
var="instance_type=t2.micro"
```

Assigning Values to Root Module Variables

- In variable definitions (.tfvars) files, either specified on the command line or automatically loaded.

```
terraform apply -var-file="testing.tfvars"
```

- As environment variables.

```
export TF_VAR_image_id=ami-abc123
terraform plan
```

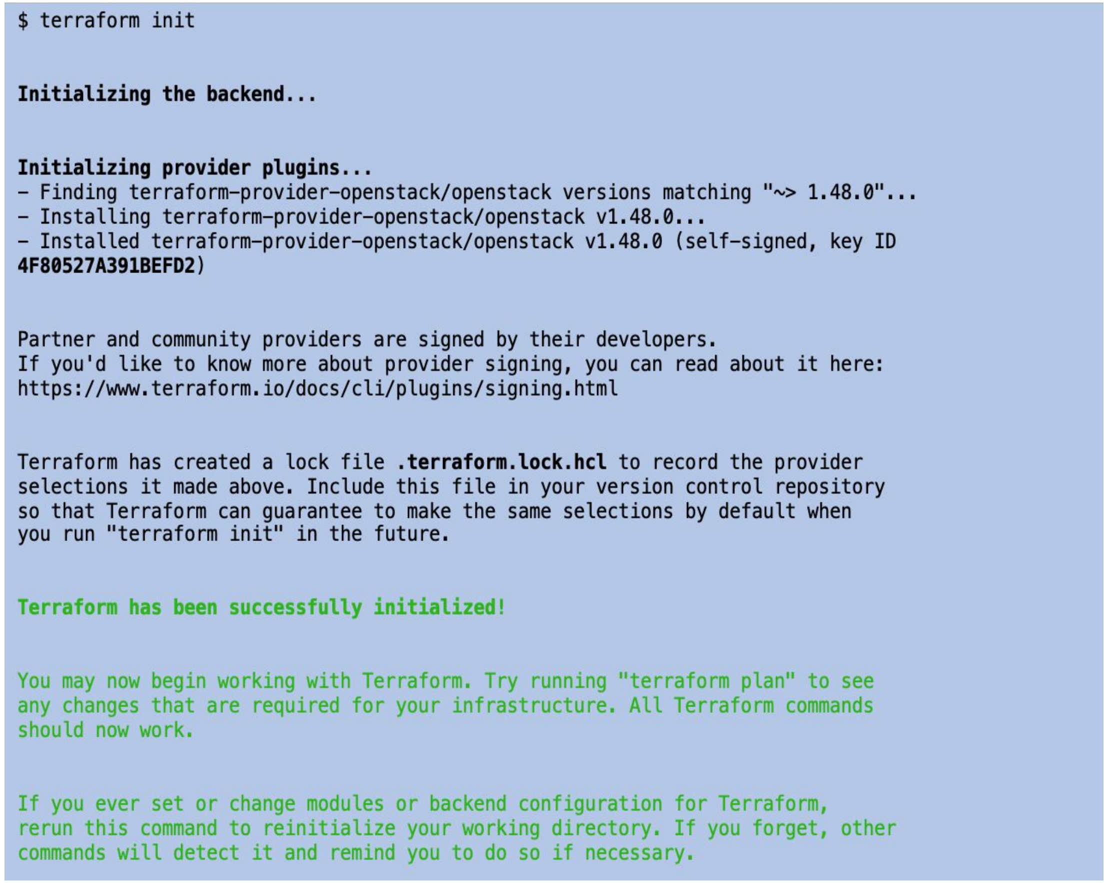
Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use. Output values are similar to return values in programming languages.

Output Values - Usage

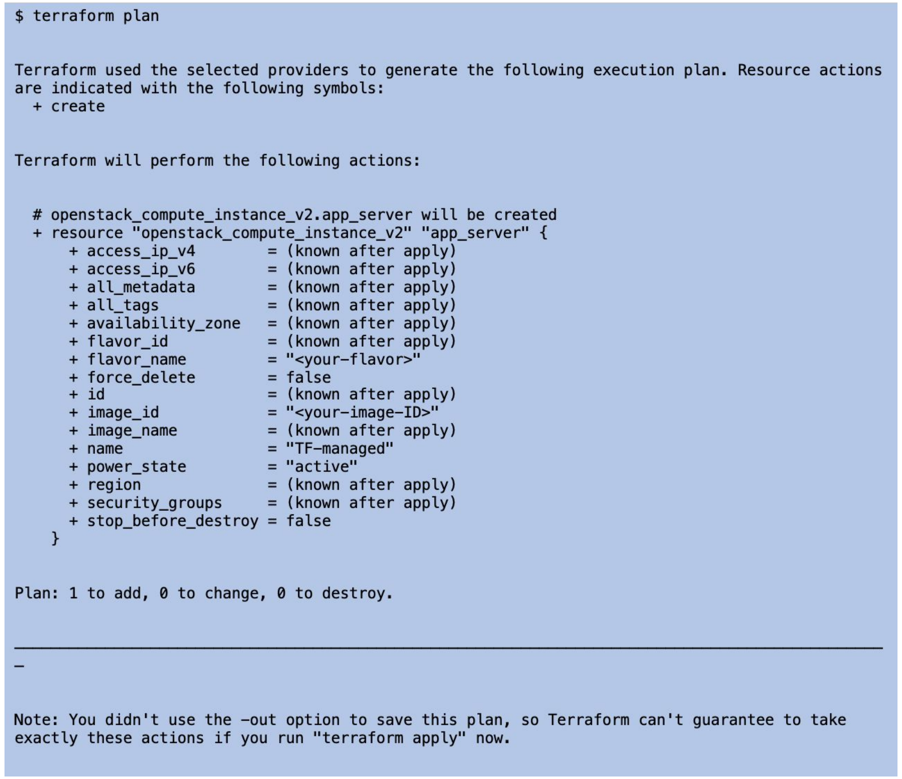
- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.
- When using remote state, root module outputs can be accessed by other configurations via a ***terraform_remote_state*** data source.

Terraform Commands - Init



```
$ terraform init


Initializing the backend...


Initializing provider plugins...
- Finding terraform-provider-openstack/openstack versions matching "~> 1.48.0"...
- Installing terraform-provider-openstack/openstack v1.48.0...
- Installed terraform-provider-openstack/openstack v1.48.0 (self-signed, key ID
4F80527A391BEFD2)


Partner and community providers are signed by their developers.
If you'd like to know more about provider signing, you can read about it here:
https://www.terraform.io/docs/cli/plugins/signing.html


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!


You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.


If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Terraform Commands - Plan



```
$ terraform plan


Terraform used the selected providers to generate the following execution plan. Resource actions
are indicated with the following symbols:
  + create


Terraform will perform the following actions:


  # openstack_compute_instance_v2.app_server will be created
  + resource "openstack_compute_instance_v2" "app_server" {
      + access_ip_v4        = (known after apply)
      + access_ip_v6        = (known after apply)
      + all_metadata        = (known after apply)
      + all_tags            = (known after apply)
      + availability_zone   = (known after apply)
      + flavor_id           = (known after apply)
      + flavor_name         = "<your-flavor>"
      + force_delete        = false
      + id                  = (known after apply)
      + image_id            = "<your-image-ID>"
      + image_name          = (known after apply)
      + name                = "TF-managed"
      + power_state         = "active"
      + region              = (known after apply)
      + security_groups     = (known after apply)
      + stop_before_destroy = false
    }


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


────────────────────────────────────────────────────────────────────────────────────────────────
─


Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take
exactly these actions if you run "terraform apply" now.
```

Terraform Commands - Apply

```
terraform apply


Terraform used the selected providers to generate the following execution plan. Resource actions
are indicated with the following symbols:
  + create


Terraform will perform the following actions:


  # openstack_compute_instance_v2.app_server will be created
  + resource "openstack_compute_instance_v2" "app_server" {
      + access_ip_v4        = (known after apply)
      + access_ip_v6        = (known after apply)
      + all_metadata        = (known after apply)
      + all_tags            = (known after apply)
      + availability_zone   = (known after apply)
      + flavor_id           = (known after apply)
      + flavor_name         = "<your-flavor>"
      + force_delete        = false
      + id                  = (known after apply)
      + image_id            = "<your-image-ID>"
      + image_name          = (known after apply)
      + name                = "TF-managed"
      + power_state         = "active"
      + region              = (known after apply)
      + security_groups     = (known after apply)
      + stop_before_destroy = false
    }


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:
```