Objectives

Upon completion of this course, you should be able to:

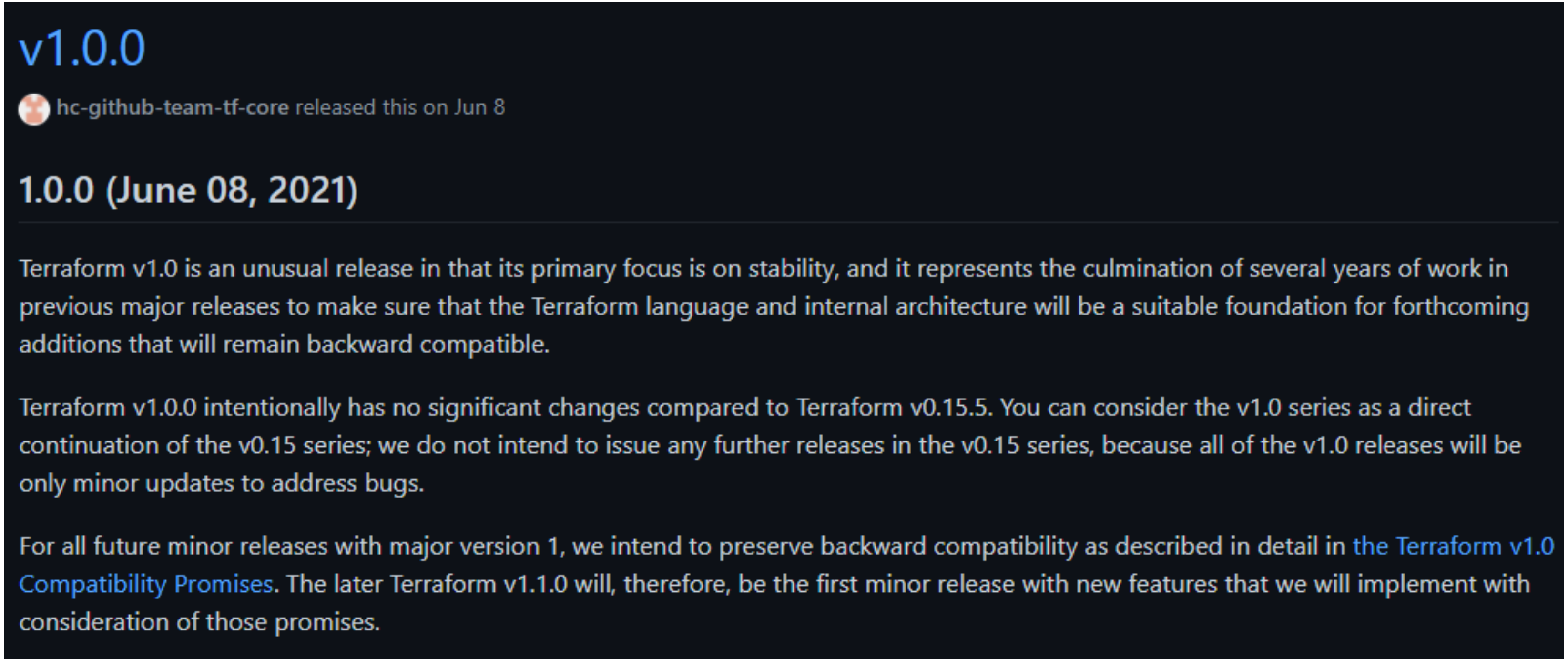
- Read and build Terraform projects
- Utilize reusable Terraform patterns
- Leverage security best practices for Terraform

Let’s Get to Know One Another

Tell me about you:
- Name
- Current role
- How long you’ve been at the company
- What’s one thing you’re hoping to get out of this course?

I’ll tell you a little about me…

Terraform Versioning



v1.0.0

hc-github-team-tf-core released this on Jun 8

1.0.0 (June 08, 2021)

Terraform v1.0 is an unusual release in that its primary focus is on stability, and it represents the culmination of several years of work in previous major releases to make sure that the Terraform language and internal architecture will be a suitable foundation for forthcoming additions that will remain backward compatible.

Terraform v1.0.0 intentionally has no significant changes compared to Terraform v0.15.5. You can consider the v1.0 series as a direct continuation of the v0.15 series; we do not intend to issue any further releases in the v0.15 series, because all of the v1.0 releases will be only minor updates to address bugs.

For all future minor releases with major version 1, we intend to preserve backward compatibility as described in detail in the Terraform v1.0 Compatibility Promises. The later Terraform v1.1.0 will, therefore, be the first minor release with new features that we will implement with consideration of those promises.

Terraform Versioning

Semantic Versioning:

- Built around the concept of patch, minor and major change tracking
- Correspond to 3 different types of possible change to the framework
- Format:

major # . *minor #* . *patch #*

Patch Change

- Rightmost digit in version number incremented
- Usually implies a bug fix
- When upgrading, represents relatively low risk

Minor Change

- Middle digit in version number incremented
- More significant change or new functionality added
- However, should remain backward compatible on upgrade

Major Change

- Leftmost digit in version number incremented
- Represents most significant type of change
- Potentially incompatible with previous versions

Terraform Versioning

- Previously, semantic versioning worked a little differently for Terraform
- Prior to 1.0.0, a change to the “minor” number looked more like “major”
- We’ve come a long way…v0.1.0 (2014) to v1.2.2 (June 2022)

https://www.hashicorp.com/blog/announcing-hashicorp-terraform-1-0-general-availability

Other Good Sources for Learning

- Terraform has been around a while now, and is popular, so there are plenty of other sources for learning
  - Hashicorp’s new learning portal: https://learn.hashicorp.com/terraform
  - Terraform repository for seeing the state of open issues: https://github.com/hashicorp/terraform/issues
  - Official Terraform docs: https://www.terraform.io/docs/index.html
  - Terraform registry for drilldown into provider-specific docs: https://registry.terraform.io/browse/providers

What is Terraform?

- “infrastructure as code”
- *declarative* domain-specific language
    - what is declarative?
- used to describe *idempotent* resource configurations, typically in cloud infrastructure
- according to Hashicorp:
    - *Terraform enables you to safely and predictably create, change, and improve infrastructure. It is an open-source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned*

What is Terraform? (cont'd)

- open source CLI tool for *infrastructure automation*
- utilizes plugin architecture
  - extensible to any environment, tool, or framework and works primarily by making API calls to those environments, tools, or frameworks
- detects implicit dependencies between resources and automatically creates a dependency graph
- builds in dependency order and automatically performs activities in parallel where possible
  - ...sequentially for dependent resources

Why Use Terraform?

- readable
- repeatable
- certainty (i.e., no confusion about what will happen)
- standardized environments
- provision quickly
- disaster recovery

What Does Terraform (HCL) Look Like?

```
resource “aws_instance” “web” {
  ami           = “ami-19827362728”
  instance_type = “t2.micro”

  tags = {
    Name = “my-first-instance”
  }
}
```

Exercise Prep: Let's Get Set Up

1. We will create an isolated (and consistent) development and execution environment where you can run Terraform

2. You will receive your student alias, access key, secret key, and console password (either via e-mail or Zoom chat)

3. Let's get these working in your development environment–this will allow you to create things and verify they exist in AWS

4. Access to your instructional repository: https://github.com/KernelGamut32/terraform-fundamentals-public/tree/main/exercises/0-setup

Format of a Terraform Project

Develop Intelligence

example-terraform-project

- .terraform (more on this directory later)
- main.tf
- outputs.tf
- variables.tf
- terraform.tfvars
- others.auto.tfvars

main.tf, outputs.tf, variables.tf $\rightarrow$ *.tf files get merged at runtime

terraform.tfvars, others.auto.tfvars $\rightarrow$ terraform.tfvars and *.auto.tfvars files get merged at runtime

Terraform Project Structure

Develop Intelligence

▷ examples	isolated units of reference implementation for modules
▷ modules	module definitions – can include subfolders
▽ environments	definitions of environment deployments
▷ staging	
▷ production	
▷ test	automated tests – e.g., terratest

Hashicorp Configuration Language (HCL)

- The goal of HCL is to build a structured configuration language that is both human and machine friendly for use with command-line tools, but specifically targeted towards DevOps tools, servers, etc.

- Fully JSON compatible

- Made up of **stanzas** or **blocks**, which roughly equate to JSON objects. Each stanza/block maps to an object type as defined by **Terraform providers** (we'll talk more about providers later)

- https://github.com/hashicorp/hcl

Terraform Project Content Types

`*.tf, *.tf.json`

- HCL or JSON
- these files define your declarative infrastructure and resources

`*.tfstate`

- JSON files that store state, reference to resources
- created and maintained by terraform

`terraform.tfvars, terraform.tfvars.json` and/or **`*.auto.tfvars, *.auto.tfvars.json`**

- HCL or JSON
- variable definitions in bulk
- (more to come on setting variable values at runtime)

Resources

- **`*.tf`** files contain your **HCL declarative** definitions

```
resource “aws_instance” “web” {
  ami           = “ami-19827362728”
  instance_type = “t2.micro”

  tags {
    Name = “my-first-instance”
  }
}
```

- most **blocks** in your HCL represent a **resource** to be created/maintained by Terraform

Resources

- *resources* are key elements and captured as top-level objects (stanzas) in Terraform configuration files
- each resource stanza indicates the intent to *idempotently* create that resource
- body of resource contains configuration of attributes of that resource
- each provider (e.g., AWS, Azure, etc.) provides its own set of resources and defines the configuration attributes
- when a resource is created by Terraform, it's tracked in Terraform *state*
- resources can refer to attributes of other resources, creating implicit dependencies
    - dependencies trigger sequential creation

Terraform Commands and the CLI

- The CLI is how you'll most often use terraform

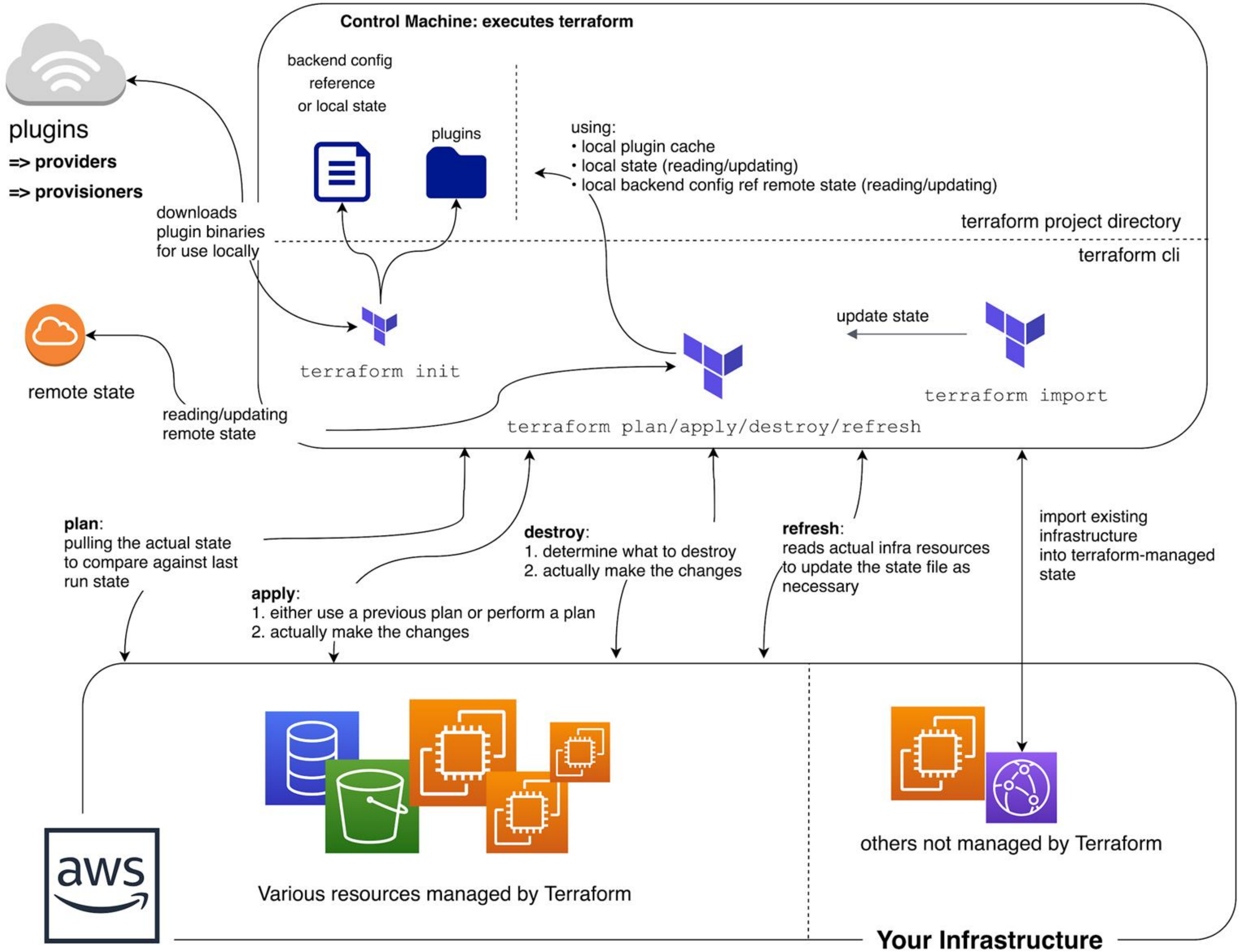
```
terraform init …
terraform plan …
terraform apply …
```

- And plenty more: `terraform --help` or https://www.terraform.io/docs/commands/index.html

- Third-party SDKs also available for running and interacting with Terraform (e.g., `scalr`, `terragrunt`, `terratest`)

Big picture look at

Terraform Command Flow



Control Machine: executes terraform
backend config reference or local state
plugins
using:
• local plugin cache
• local state (reading/updating)
• local backend config ref remote state (reading/updating)
terraform project directory
terraform cli
plugins
=> providers
=> provisioners
downloads plugin binaries for use locally
terraform init
update state
terraform import
remote state
reading/updating remote state
terraform plan/apply/destroy/refresh
plan: pulling the actual state to compare against last run state
apply: 1. either use a previous plan or perform a plan 2. actually make the changes
destroy: 1. determine what to destroy 2. actually make the changes
refresh: reads actual infra resources to update the state file as necessary
import existing infrastructure into terraform-managed state
aws
Various resources managed by Terraform
others not managed by Terraform
Your Infrastructure

`terraform init`

- a special command, run before other commands/operations
- what does it do?
  - downloads required provider packages
  - downloads modules referenced in the HCL (more on modules later)
  - initializes state
    - local state: ensuring local state file(s) exist
    - remote state: more complex initialization (more on remote state later)
  - basic syntax check

- idempotent

- remember the **`.terraform`** directory?
  - **`init`** downloads the provider packages and modules to this directory
  - also, where state files live

Exercise 1: First Terraform Project

Exercise 1

Extending Your Project

- Input Variables
- Locals
- Data Sources
- Provisioners
  - `remote-exec`
  - `local-exec`
  - `null_resource`

Input Variables

- enable interchangeable values to be stored centrally and referenced single or multiple times
- similar to variables in other languages
- declared in **`variable`** stanzas
- parsed first
- cannot interpolate or reference other variables
- allow for default values
- optionally specify value type, e.g.,
    - **`List, Map, String`**

Input Variables

- Input variable definitions support the following
  - default – provides default value if not specified; makes optional
  - type – type of value accepted for the variable
  - description – string description/documentation
  - validation – block for defining validation rules for input
  - sensitive – true or false; limits output as part of TF operations (plan or apply)

Example Variable Definition

```
variable “instance_size” {
   default     = “t2.micro”
   type        = string # changed in 0.12
   description = “Size of EC2 instance”
}
```

Example Variable Definition

```
variable “student_alias” {
  type          = string
  description = “Your student alias”
  validation {
    condition     = trimprefix(var.student_alias, “test”) == var.student_alias
    error_message = “Please do not use test aliases with this deployment.”
  }
}
```

Locals

- mutable values that allow for interpolation and inference
- **CAN** reference variables and other locals
- **CAN'T** be set via arguments from the command line
- use them when a value is used in many places in your code and that value is likely to change
- don't overuse them or your code can be difficult to read

Local Definitions

```
locals {
   one       = “1”
   twelve    = “${local.one}2”
   onetwelve = “${local.one}${local.twelve}”
}
```

Data Sources

- logical references to data objects stored externally to the `tfstate` file
- allows you to reference resources not created by Terraform
- examples
    - current default region in AWS CLI
    - AMI ID search
    - AWS ARN lookup
    - AWS VPC CIDR range

Data Source Example: AWS AMI Lookup

Develop Intelligence

```
data "aws_ami" "latest-ubuntu" {
  most_recent = true
  owners = ["099720109477"]

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Provisioners

- allow you to run commands during instance provisioning that are run on create, recreate, or taint correction (explained later), but not every time **`terraform apply`** is run
- ties custom logic to idempotent resources
- types
  - local
  - remote
  - **`chef`**
- connectors
  - SSH
  - WinRM

Provisioner Example: `local-exec`

```
resource "aws_instance" "web" {
  ami           = "ami-19827362728"
  instance_type = "t2.micro"

  tags {
    Name = "my-first-instance"
  }

  provisioner "local-exec" {
    command = "echo 'created instance'"
  }
}
```

Provisioner Example: remote-exec

```
resource “aws_instance” “web” {
  ...

  provisioner “remote-exec” {
    inline = [
"sudo sed -i
    's/^PasswordAuthentication.*/PasswordAuthentication yes/'
    /etc/ssh/sshd_config",
"sudo service sshd restart",
"wget https://repo.anaconda.com/Anaconda3-Linux-x86_64.sh",
"sh Anaconda3-Linux-x86_64.sh -b"
             ]
  }
}
```

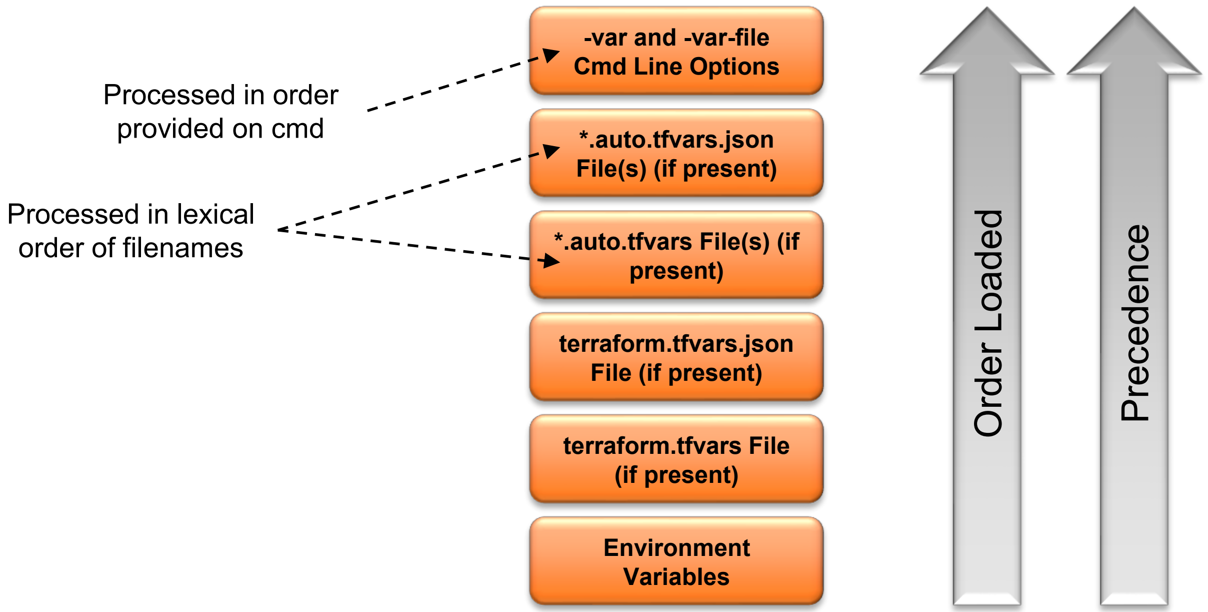
Provisioner example: `null_resource`

```
resource “null_resource” “first-tf-run” {
  provisioner “local-exec” {
    command = “echo ‘this will run on first tf
apply’”
  }
}
```

Providing Values for Input Variables

- Multiple options
  - using environment variables (prefixed with “TF_VAR_”)
  - defining inputs in a terraform.tfvars file
  - defining inputs in a terraform.tfvars.json file
  - defining inputs in one or more *.auto.tfvars files
  - defining inputs in one or more *.auto.tfvars.json files
  - -var and -var-file options on the command-line

Providing Values for Input Variables


-var and -var-file Cmd Line Options
*.auto.tfvars.json File(s) (if present)
*.auto.tfvars File(s) (if present)
terraform.tfvars.json File (if present)
terraform.tfvars File (if present)
Environment Variables
Processed in order provided on cmd
Processed in lexical order of filenames
Order Loaded
Precedence

Providing Values for Input Variables

- primarily used when executing Terraform via CLI
- not really used with Terraform Enterprise
- can “push” those variables + values to Enterprise (in files)
- but manage from “Variables” section of the environment

Exercise 2: Using Variables

Exercise 2

How Terraform Works

- state and how to query it
- computing plans
- executing plans (`terraform apply`)

State

- stores information about resources that are created by Terraform
  - also includes values computed by the provider APIs

- local file
  - `.tfstate`

- or backends are also available…

Backends

- determines how state is loaded and how operations like **`apply`** are executed

- enables non-local file state storage, remote execution, etc.

- why use a backend?
  - can store their state remotely and protect it to prevent corruption
    - some backends, e.g., *Terraform Cloud* automatically store all revisions
  - keep sensitive information off local disk
  - remote operations
    - apply can take a *LONG* time for large infrastructures

Backends (cont'd)

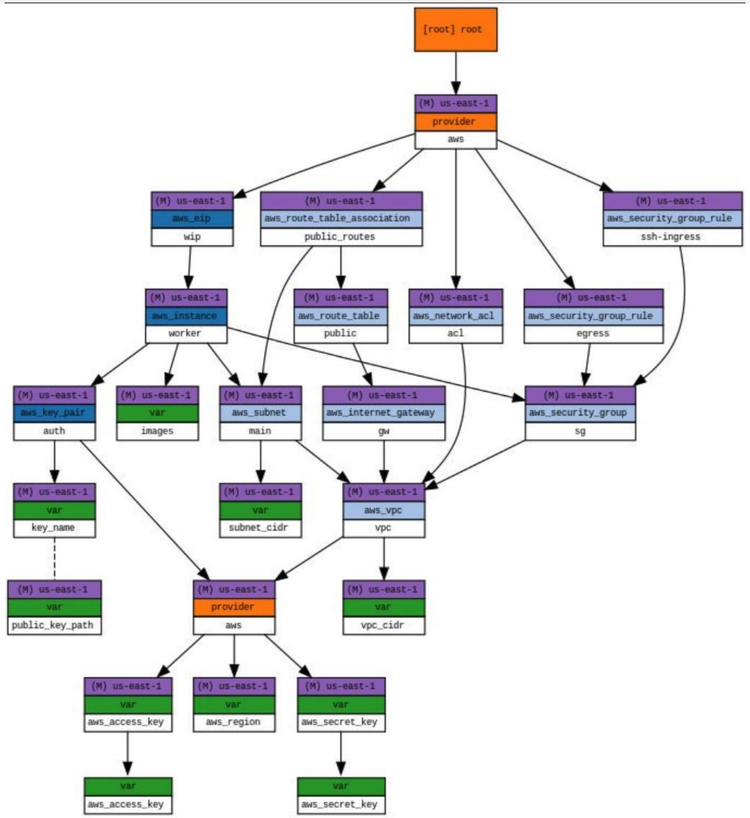
- examples
  - S3
  - swift
  - http
  - Terraform Enterprise
  - etc.

How to Query or See the Current State

- CLI
  - `terraform show [-json]`
    https://www.terraform.io/docs/commands/show.html

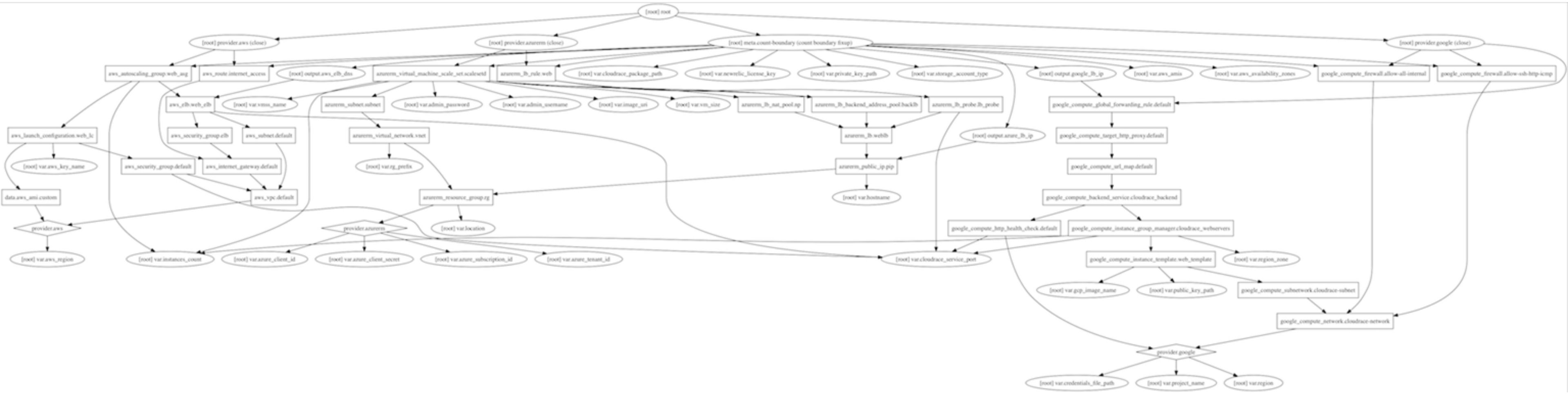
- Remote State Data Type
  https://www.terraform.io/docs/providers/terraform/d/remote_state.html

Visualizing Graph Outputs



[root] root
(M) us-east-1 provider aws
(M) us-east-1 aws_eip wip
(M) us-east-1 aws_route_table_association public_routes
(M) us-east-1 aws_security_group_rule ssh-ingress
(M) us-east-1 aws_instance worker
(M) us-east-1 aws_route_table public
(M) us-east-1 aws_network_acl acl
(M) us-east-1 aws_security_group_rule egress
(M) us-east-1 aws_key_pair auth
(M) us-east-1 var images
(M) us-east-1 aws_subnet main
(M) us-east-1 aws_internet_gateway gw
(M) us-east-1 aws_security_group sg
(M) us-east-1 var key_name
(M) us-east-1 var subnet_cidr
(M) us-east-1 aws_vpc vpc
(M) us-east-1 var public_key_path
(M) us-east-1 provider aws
(M) us-east-1 var vpc_cidr
(M) us-east-1 var aws_access_key
(M) us-east-1 var aws_region
(M) us-east-1 var aws_secret_key
var aws_access_key
var aws_secret_key

Visualizing Graph Outputs



[root] root
[root] provider.aws (close)
[root] provider.azurerm (close)
[root] meta.count-boundary (count boundary fixup)
[root] provider.google (close)
aws_autoscaling_group.web_asg
aws_route.internet_access
[root] output.aws_elb_dns
azurerm_virtual_machine_scale_set.scalesetd
azurerm_lb_rule.web
[root] var.cloudrace_package_path
[root] var.newrelic_license_key
[root] var.private_key_path
[root] var.storage_account_type
[root] output.google_lb_ip
[root] var.aws_amis
[root] var.aws_availability_zones
google_compute_firewall.allow-all-internal
google_compute_firewall.allow-ssh-http-icmp
aws_elb.web_elb
[root] var.vmss_name
azurerm_subnet.subnet
[root] var.admin_password
[root] var.admin_username
[root] var.image_uri
[root] var.vm_size
azurerm_lb_nat_pool.np
azurerm_lb_backend_address_pool.backlb
azurerm_lb_probe.lb_probe
google_compute_global_forwarding_rule.default
aws_launch_configuration.web_lc
aws_security_group.elb
aws_subnet.default
azurerm_virtual_network.vnet
azurerm_lb.weblb
[root] output.azure_lb_ip
google_compute_target_http_proxy.default
[root] var.aws_key_name
aws_security_group.default
aws_internet_gateway.default
[root] var.rg_prefix
azurerm_public_ip.pip
google_compute_url_map.default
data.aws_ami.custom
aws_vpc.default
azurerm_resource_group.rg
[root] var.hostname
google_compute_backend_service.cloudrace_backend
provider.aws
provider.azurerm
[root] var.location
google_compute_http_health_check.default
google_compute_instance_group_manager.cloudrace_webservers
[root] var.aws_region
[root] var.instances_count
[root] var.azure_client_id
[root] var.azure_client_secret
[root] var.azure_subscription_id
[root] var.azure_tenant_id
[root] var.cloudrace_service_port
google_compute_instance_template.web_template
[root] var.region_zone
[root] var.gcp_image_name
[root] var.public_key_path
google_compute_subnetwork.cloudrace-subnet
google_compute_network.cloudrace-network
provider.google
[root] var.credentials_file_path
[root] var.project_name
[root] var.region

Executing Plans: `terraform apply`

- What does **`terraform apply`** do?
  - syntax check
  - check for init
  - refresh state
  - execute plan
  - ask for input
  - execute changes

Exercise 3: Plans and Applies

Exercise 3

Providers

- responsible for understanding API interactions and exposing resources
- Hashicorp helps companies create providers to be added to ecosystem
- declared in HCL config files as a **`provider`** stanza
- each Terraform project can have multiple providers, even of the same type
- describes resources, their inputs, outputs, and the logic to create and change them
- many options
    - AWS, GCP, Azure, and many many others
    - providers available for non-infra services as well such as gmail, MySQL, and Pagerduty

The AWS Provider

- provider documentation
    - https://www.terraform.io/docs/providers/aws/index.html

- HUGE amount of resources

- something like 8 resources per service on average

Configuring the Provider

```
provider “aws” {
   region     = “us-west-1”
   access_key = “[your access key]”
   secret_key = “[your secret access key]”
}
```

Reuse Patterns in Terraform

- **Workspaces**: separate state files for the same HCL
- **Outputs**: automated use of terraform-managed resources
- **Modules**: packaged HCL for reuse

Workspaces

- Workspaces allow you to use the same configuration (HCL/project) for multiple states
  - example: AWS Developer VPCs–each developer could have an identical environment as defined by the configuration, but each managed by a different state by way of separate workspaces

- nothing more than separately-named state files

- both local and remote state backends support workspaces

Output Variables

- *inputs* to a Terraform config are declared with variables stanzas
- *outputs* are declared with a special output stanza
- can be referenced through the modules interface or the CLI

Output Variables

- Output variable definitions support the following
    - value – value to be returned as output
    - description – string description/documentation
    - sensitive – true or false; limits output as part of TF operations (plan or apply)

Output Definition

```
output “instance_public_ip” {
  value = aws_instance.web.public_ip
}
```

Modules

- *modules* are a critically important concept in Terraform
- basically, every Terraform working directory, as long as it has variable stanzas, is a module
- this allows developers to compose reusable blocks of configuration and reference them with module stanzas

Modules

- allow for modularized configuration (create separate modules for different parts of configuration), aka module composition
- every project has at least one module (the “root” module), but root can have a tree of children
- child modules have input variables passed in from parent module
- modules can be defined by configuration files in local filesystem or remote source

Modules

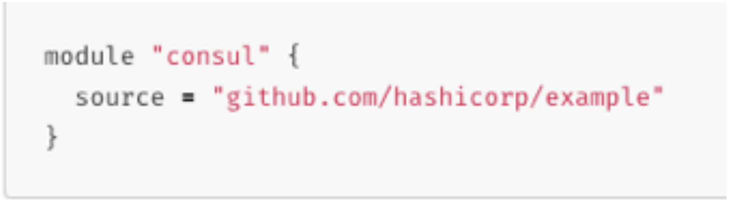
- can publish modules in Terraform registry to make them easy to find
  - **source** attribute identifies location of module, e.g.,

```
module “webserver” {
  source = “./webserver” # module in this dir
  instance_type = “t2.micro”
}
```

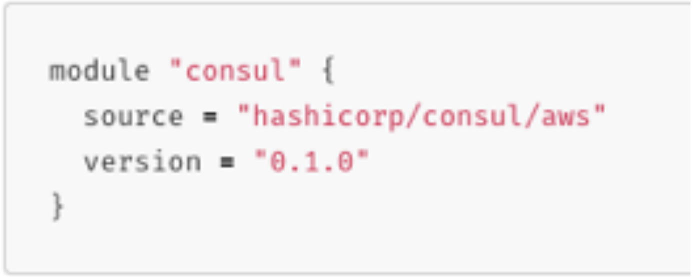
- most attributes of a module are input variables passed in from parent
- module’s outputs can be accessed and used by parent (and passed to other child modules of the parent)

Module Sources

- Terraform allows the user to pull modules from various locations
  - local paths
  - Github
  - Terraform Registry
  - Bitbucket
  - HTTP
  - S3 Buckets



```
module "consul" {
  source = "github.com/hashicorp/example"
}
```



```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

- More info
  - https://www.terraform.io/docs/modules/sources.html

Module Example

```
variable “thing” {
  type = string
}

resource null_resource “null” {
  provisioner local-exec {
    command = "echo ${var.thing}"
  }
}
```

Using the Module

```
module “my_module” “printer” {
   source = “./my_module”

   # this is a variable passed into module
   thing = “this should be printed”
}
```

Exercise 4: Querying State

Exercise 4

Exercise 5: Interacting with Providers

Exercise 5

Exercise 6: Modules

Exercise 6

Error Handling and Debugging

- Most errors fall into one of four types
  - Process Errors
  - Syntax Errors
  - Validation Errors
  - Passthrough Errors

Process Errors

- errors due to process not being followed
- e.g.,
  - running **`apply`** before **`init`**
  - variables not fully populated

Syntax Errors

- caused by an error in syntax, e.g.,
  - HCL codebase syntax or parameter errors
  - incorrect usage of built-in functions
  - type errors

Validation Errors

- preliminary validation built into provider occuring before plan
- usually more detailed

Provider Errors / Passthrough

- errors received from provider or third-party API while in process of refresh, plan, apply, etc.
    - usually most difficult to troubleshoot
    - requires knowledge of provider’s tech (e.g., AWS)

Troubleshooting Commands

- **`terraform validate`**
  - performs a syntax check on all terraform files in the directory
  - displays an error if any of the files doesn't validate
  - does *NOT* check formatting
  - what does it check…?

`terraform validate`

- invalid HCL syntax (e.g., missing quote or equal sign)
- invalid HCL references (e.g., variable name or attribute which doesn't exist)
- same provider declared multiple times
- same module declared multiple times
- same resource declared multiple times
- invalid module name
- interpolation used in places where it's unsupported (e.g., variable, **`depends_on`**, **`module.source`**, **`provider`**)
- missing value for a variable (none of **`-var foo=...`** flag, **`-var-file=foo.vars`** flag, **`TF_VAR_foo`** environment variable, terraform.tfvars, or default value in the configuration)

Troubleshooting: `terraform fmt`

- rewrites Terraform files in a canonical format/style
- by default, scans the current directory for configuration files
    - if the dir argument is provided then it will scan that given directory instead
    - if dir is a single dash (-) then **`fmt`** will read from standard input

Troubleshooting: `terraform graph`

- generates a visual representation of either a configuration or execution plan
  - output is in DOT format, which can be used by GraphViz to generate charts: https://www.terraform.io/docs/internals/graph.html

- e.g.,

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

Troubleshooting: `terraform console`

- creates an interactive console for testing interpolations
  - similar to running the Python interpreter in interactive mode
- great for testing complex conditionals

Exercise 7: Error Handling, Troubleshooting

Exercise 7

Interpolation

- embedded within strings in Terraform, whether you're using the HCL or JSON, you can interpolate other values.
  - These interpolations are wrapped in **`${...}`**, such as **`${var.foo}`**
- allows you to reference variables, attributes of resources, call functions, etc.
- simple math like
  - **`${count.index + 1}`**
- allows for conditional statements
- https://www.terraform.io/docs/language/expressions/strings.html

Built-in Functions

- built-in functions:
  - Terraform ships with built-in functions
  - called with the syntax **`name(arg, arg2, ...)`**
  - e.g., to read a file:

    ```
    ${file("path.txt")}
    ```
  - https://www.terraform.io/docs/language/functions/index.html

Conditionals

- interpolations may contain conditionals to branch on the final value

- syntax

```
    CONDITION ? TRUEVAL : FALSEVAL
```

Primitive Data Types

- `string`
  - use the var prefix followed by the variable name
  - e.g., **`${var.foo}`** is how you would use the variable in HCL for interpolation or reference
- `number`
  - can be referenced as a number, so in arithmetic for example

    ```
    ${var.foo + 1}
    ```
- `bool`
  - can be referenced as a boolean in logic, so something like

    ```
    ${var.foo == true ? "foo is true" : "foo is false"}
    ```

Extended Data Types

- **`list(<type>)`**
  - ordered list of things, i.e., array
    - e.g., **`${var.subnets}`** would get the value of the subnets *list*
    - you can also return list elements by index: **`${var.subnets[0]}`**
- **`set(<type>)`**
  - similar to a list, but: requires a type, unique values, no ordering
- **`map(<type>)`**
  - a collection of values where each is identified by a string
  - e.g., **`${var.amis["us-east-1"]}`** would get the value of the **`us-east-1`** key within the **`amis`** map variable

Extended Data Types (cont’d)

- `object({ <attr name> = <type>, … })`
  - like many other language object types, with properties containing other values
- `tuple([<type>, …])`
  - very similar to a list, mixed strictly defined typed list of things

Count Parameter

- resources can be duplicated or conditionally created via the count parameter

```
resource “aws_instance” “web” {
  count         = 2
  ami           = “${var.ami}“
  instance_type = “${var.instance_type}”

  tags {
    Name = “web-${count.index}”
  }
}
```

Data Reference

- attributes of current resource
  - syntax is **`self.ATTRIBUTE`**
  - e.g., **`${self.private_ip}`** interpolates resource's private IP address

Data Reference (cont’d)

- attributes of other resources
  - syntax is **`TYPE.NAME.ATTRIBUTE`**
  - **`${aws_instance.web.id}`**
    - interpolate ID attribute from the **`aws_instance`** resource **`web`**
    - if resource has a count attribute set, you can access individual attributes with a zero-based index, such as **`${aws_instance.web[0].id}`**
    - or use the splat syntax to get a list of all the attributes: **`${aws_instance.web[*].id}`**

Data Sources and Reference

- attributes of a data source
  - **`data.TYPE.NAME.ATTRIBUTE`**
  - **`${data.aws_ami.ubuntu.id}`**
    - interpolate **`id`** attribute from the **`aws_ami`** data source **`ubuntu`**
    - if data source has a **`count`** attribute set, access individual attributes with a zero-based index, e.g., **`${data.aws_subnet.example[0].cidr_block}`**
    - or use the splat syntax to get a list of all the attributes: **`${data.aws_subnet.example[*].cidr_block}`**

Data Sources and Reference (cont'd)

- Referencing values output from another module
  - `module.MODULE_NAME.MODULE_OUTPUT_NAME`

Resource Example w/Conditional

```
resource “aws_instance” “web” {
   ami           = “${var.ami}“
   instance_type = “${var.instance_type}”

   tags {
      Name = “${var.env == “production” ? “production-
web” : “staging-web” }”
   }
}
```

Exercise 8: Understanding & Manipulating Data/Variables

Exercise 8

Keeping Terraform in Sync with Infra

- configuration drift
  - things change!
  - Terraform can bring those things back in line naturally
- **`plan`**
  - when executing a plan, Terraform can output machine readable syntax (exit codes) that can be used to monitor for manual infra changes
  - if the infra changes, plans will suddenly detect drift and inform alarms
- **`apply`**
  - thanks to Terraform’s idempotency, corrections are natural and easy

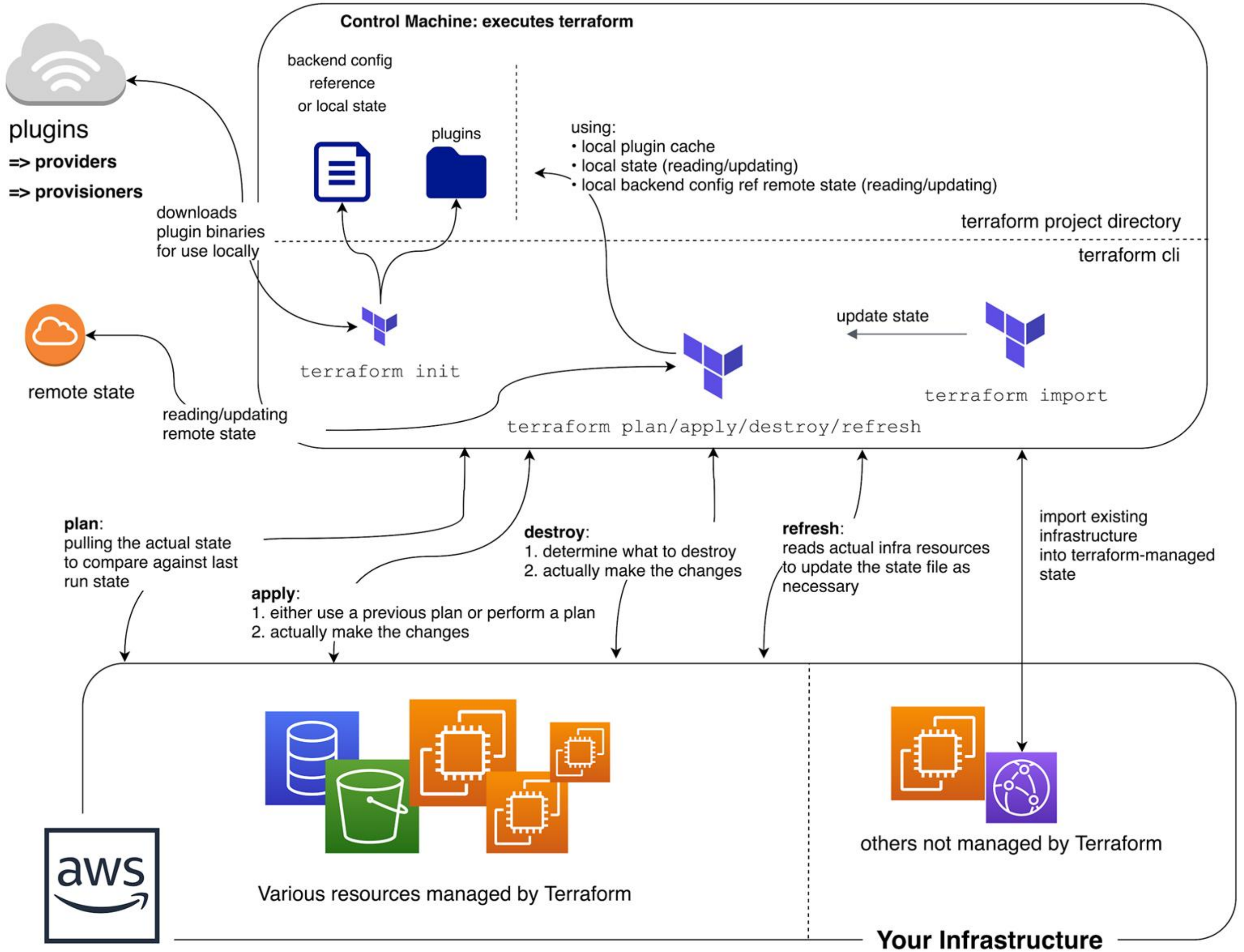
Keeping Terraform in Sync with Infra

- what if we want to keep the changes?
- you can import them
  - use **`terraform import`** to pull in the changes to the state
  - must also change the Terraform config to match any changes
  - if you have a clean plan with no planned changes, you were successful
- e.g.,

```
terraform import aws_instance.my_instance i-abcd1234
```

Big picture look at

Terraform Command Flow


Control Machine: executes terraform
backend config reference or local state
plugins
using:
• local plugin cache
• local state (reading/updating)
• local backend config ref remote state (reading/updating)
terraform project directory
terraform cli
plugins
=> providers
=> provisioners
downloads plugin binaries for use locally
terraform init
update state
terraform import
remote state
reading/updating remote state
terraform plan/apply/destroy/refresh
plan: pulling the actual state to compare against last run state
apply: 1. either use a previous plan or perform a plan 2. actually make the changes
destroy: 1. determine what to destroy 2. actually make the changes
refresh: reads actual infra resources to update the state file as necessary
import existing infrastructure into terraform-managed state
aws
Various resources managed by Terraform
others not managed by Terraform
Your Infrastructure

Exercise 9: Resource Counts and Conditionals

Exercise 9

Terraform Expressions

- can set attributes, outputs and locals to expressions
- expressions can refer to
  - literal values or complex literal values: **`true, 13, “us-west1”, [1, 2], {a:1, b:2}`**
  - resource or data source attributes: **`<RESOURCE TYPE>.<NAME>, data.<DATA TYPE>.<NAME>`**
  - type indices: **`local.list[3], local.object.attrname, local.map["keyname"]`**
  - variables: **`var.<NAME>`**
  - locals: **`local.<NAME>`**

  …

Terraform Expressions (cont'd)

- module outputs: **`module.<MODULE NAME>.<OUTPUT NAME>`**
- path variables: **`path.module, path.root, path.cwd`**
- workspace setting: **`terraform.workspace`**
- built-in functions using any of the above as arguments
  - **`max(5, 12, var.my_value)`**
- arithmetic, logical, or comparison operators combining the above
- conditional expressions: **`var.a != "" ? var.a : “default-a”`**
- string template interpolation: **`“Hello, ${var.name}!”`**
- string template directives:

```
   "Hello, %{ if var.name != "" }${var.name}%{ else }unnamed%{ endif
}!"
```

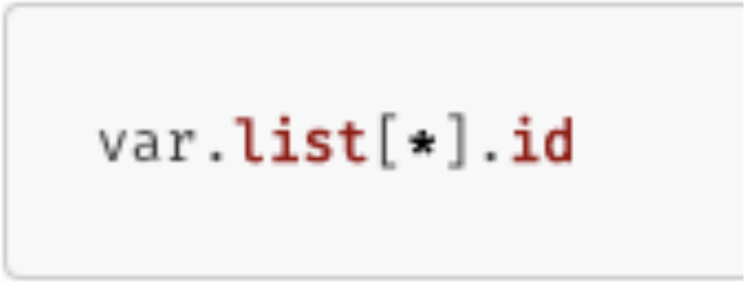
Terraform Expressions (cont'd)

- Multi-line string templates
- Looping string directive (**`for/endfor`**)

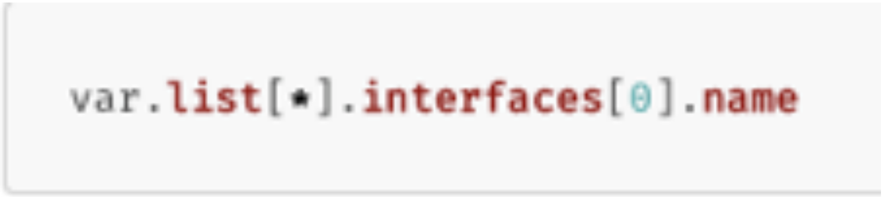
```
<<EOT
%{ for ip in aws_instance.example.*.private_ip }
server ${ip}
%{ endfor }
EOT
```

Terraform Expressions

- **Splat syntax**



```
var.list[*].id
```



```
var.list[*].interfaces[0].name
```

Terraform Expressions

- `for` expressions to convert lists/maps/tuples/objects to other lists/maps/tuples/objects

```
[for s in var.list : upper(s)]
```

```
{for s in var.list : s => upper(s)}
```

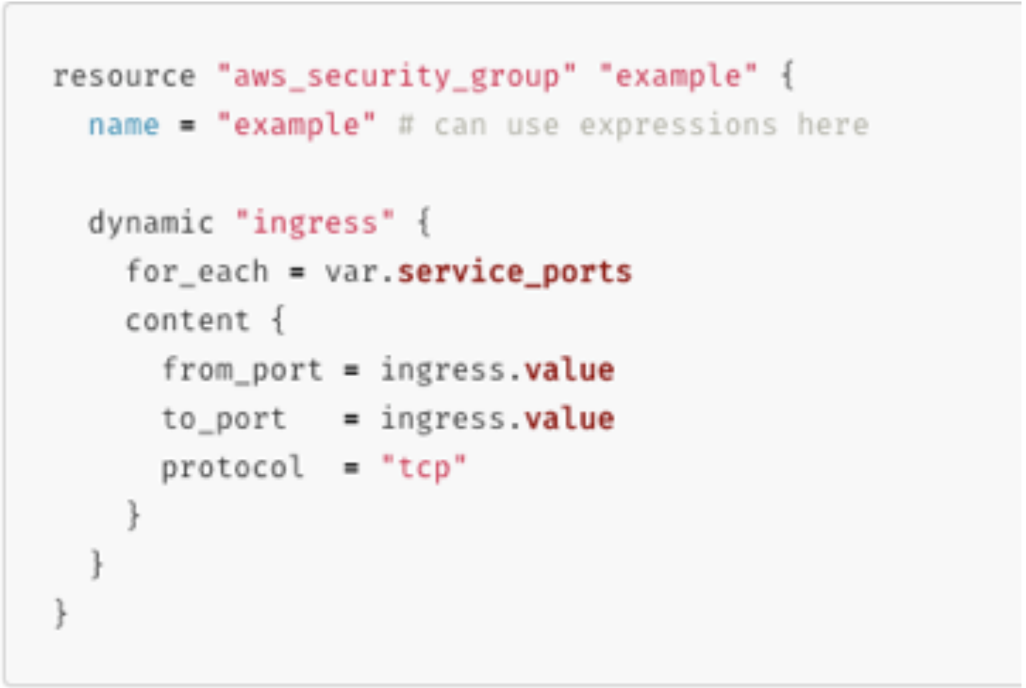
```
[for s in var.list : upper(s) if s != ""]
```

```
[for k, v in var.map : length(k) + length(v)]
```

```
{for s in var.list : substr(s, 0, 1) => s... if s != ""}
```

Terraform Expressions

- **`dynamic`** blocks



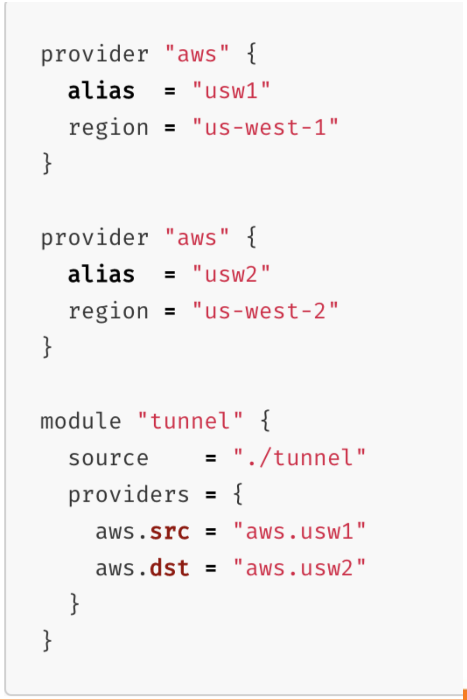
```
resource "aws_security_group" "example" {
  name = "example" # can use expressions here

  dynamic "ingress" {
    for_each = var.service_ports
    content {
      from_port = ingress.value
      to_port   = ingress.value
      protocol  = "tcp"
    }
  }
}
```

Terraform Meta-arguments

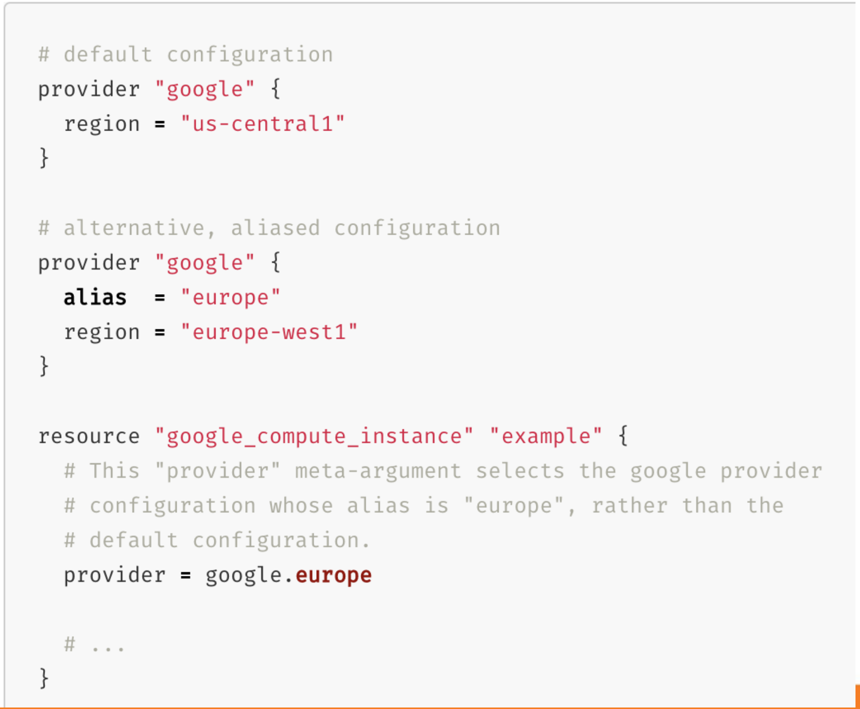
- resources, data sources, modules, and outputs can have meta-arguments (available across all types of all providers)
- modules have: `source`, `version`, `providers`
- outputs have: `depends_on`
- resources have: `depends_on`, `count`, `for_each`, `provider`, `lifecycle`, `provisioner` (`provisioner` can have `connection` inside)
- data sources have same as resources except for `lifecycle`
- `depends_on` forces a dependency on another object even if no implicit dependency by referring to an attribute of another object
- `lifecycle` controls how resources are modified when configuration changes
- `for_each` is like count except it iterates over a set (unordered list) or map; has `each.key`, `each.value` instead of `count.index` to refer to each index

Terraform Meta-arguments

- `providers` and `provider` are used when dealing with multiple providers in the same configuration



```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = "aws.usw1"
    aws.dst = "aws.usw2"
  }
}
```



```
# default configuration
provider "google" {
  region = "us-central1"
}

# alternative, aliased configuration
provider "google" {
  alias  = "europe"
  region = "europe-west1"
}

resource "google_compute_instance" "example" {
  # This "provider" meta-argument selects the google provider
  # configuration whose alias is "europe", rather than the
  # default configuration.
  provider = google.europe

  # ...
}
```

Backends

- Backends are the concept that terraform uses to store state
- Defaults to local tfstate file
- Others available:
    - S3
    - HTTP
    - Consul
    - Artifactory
    - Etcd
    - Terraform Enterprise
    - etc...

Backends: S3

Because this course is about Terraform with AWS specifically, let's talk about the S3 state backend:

- Uses a bucket and path to an object (the state file) in the bucket for a central place to store state
- Supports locking using an AWS Dynamo DB table
- See https://www.terraform.io/docs/backends/types/s3.html for more info

Exercise 10: S3 Backends

Exercise 10

Exercise 11: Running an Application in AWS

Exercise 11

Security and Terraform

Really two levels:

- Security at the IaC (Infrastructure-as-Code) level
- Secure configuration of the target resources (e.g., AWS components)

Security and Terraform

At the target resource level:

- use best practices as defined by cloud provider (e.g., AWS or Azure Well Architected Framework)
- utilize network boundaries and controls (VPC’s, subnets, security groups)
- utilize TLS and encryption at rest and in transit
- utilize API gateways and firewalls to protect at the perimeter
- mTLS is a great option for service-to-service integration
- practice the principles of “least privilege” and defense in depth

Security and Terraform

Common missteps at the target resource level:

- using default configurations not optimized for security
- not sufficiently leveraging logging and not managing sensitive data in logs
- using unencrypted data stores
- using less secure protocols for network communications
- not effectively accounting for both AuthN and AuthZ

Security and Terraform

At the IaC (Terraform) level:

- apply proper controls by environment and for state at each environment
- be aware of the 3rd party modules used and any vulnerabilities
- use variables as control points in the configuration
- shift security left – use static and dynamic scanning tools for security
- look for potential security issues in the Terraform plan

Terraform Enterprise

Terraform Enterprise

What is it?

- self-hosted distribution of Terraform Cloud
- private instance of Terraform Cloud for added security & control
- no resource limits
- additional features – audit logging, SAML SSO, etc.

Terraform Enterprise

Benefits:

- improvements in operational efficiencies
- controlling cloud costs
- reducing risk

Terraform Enterprise

Improvements in operational efficiencies:

- graphical access to workspaces and runs within
- graphical access to variables and outputs
- graphical access to remote state
- log output from each state of workflow

Terraform Enterprise

Controlling cloud costs:

- cost estimation built into workflow
- integration with Terraform Sentinel policies brings control
- policies can be used to prevent overprovisioning, integration of an approval step for expensive resources, etc.

Terraform Enterprise

Reducing risk:

- Terraform Sentinel policies help with enforcement of security best practices
- private module registry offers security for IaC IP
- facilitates team coordination
- adds audit logging and logging of approvals for traceability
- capabilities of Terraform Cloud but self-hosted and controlled

Terraform Enterprise

Integration with VCS:

- supports direct integration with source control (like GitHub)
- as configuration updates are checked in, workflow in UI able to execute

Terraform Sentinel

Terraform Sentinel

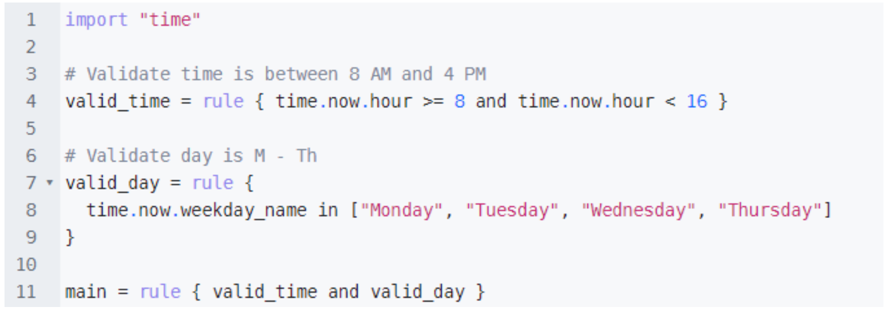
What is it?

- embedded policy-as-code framework
- integrated into Terraform Cloud (and by extension Terraform Enterprise)
- enables fine-grained, logic-based policy decisions for securing IaC

Terraform Sentinel

Process:

- define – uses a proprietary policy language (kinda Go “like”)



```
import "time"

# Validate time is between 8 AM and 4 PM
valid_time = rule { time.now.hour >= 8 and time.now.hour < 16 }

# Validate day is M - Th
valid_day = rule {
  time.now.weekday_name in ["Monday", "Tuesday", "Wednesday", "Thursday"]
}

main = rule { valid_time and valid_day }
```

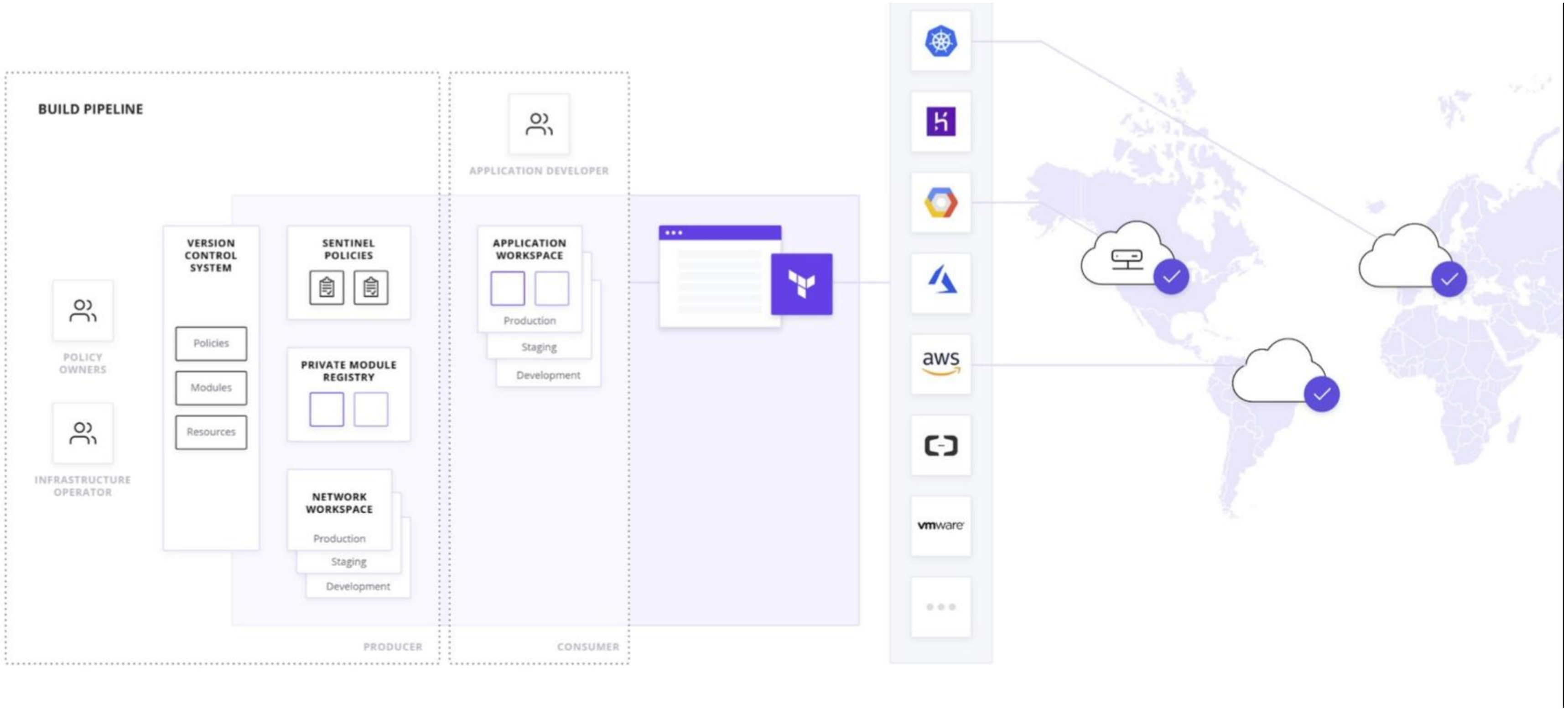
- manage – via VCS integration or policy upload through API
- enforcement – adding policy checks to runs

Terraform Sentinel

Mocking & Testing

- in Terraform Cloud, able to generate mock data for a run
- mock data can be used with Sentinel CLI to test policies prior to deployment

Terraform Enterprise – Reference Arch



Terraform Enterprise - Demo

THANK YOU

Develop Intelligence