# 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*

# Experiments

- an extension of exercises
  - we can take a closer look together as to how this applies to real scenarios in your current work
- choose your own adventure or I can give some ideas to explore
- uncovering Terraform “gotchas” and limitations
- Anything spark ideas for you along the way that you’d like to see in action?
- How can I help you get what you need to out of this course?

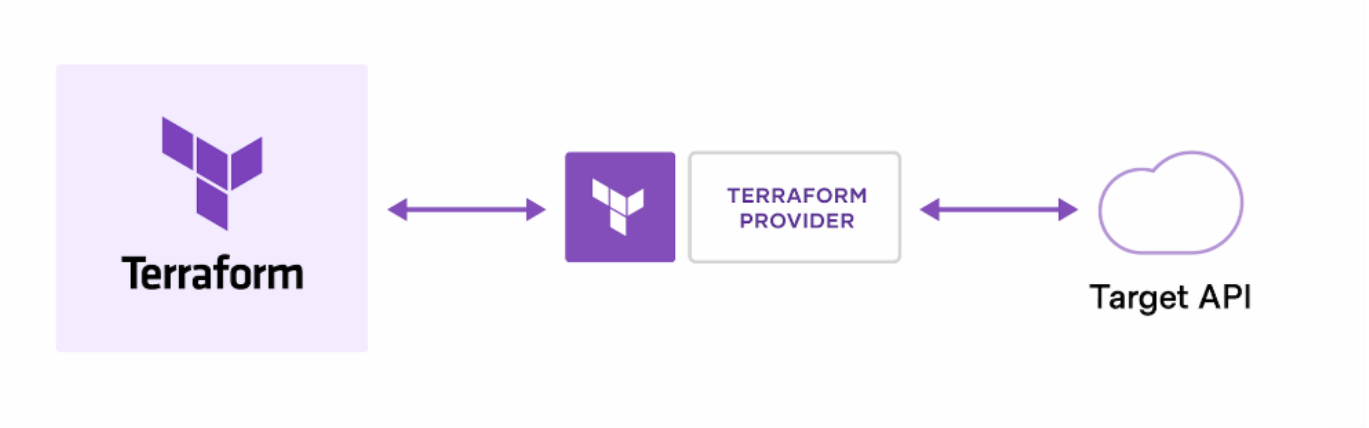
# Other Good Sources for Learning

- Terraform has been around a while now, and is pretty 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
  - Pluralsight path (if you have account): https://app.pluralsight.com/paths/skills/managing-infrastructure-with-terraform

# 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

# Provider plugin architecture



# 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”
  }
}
```

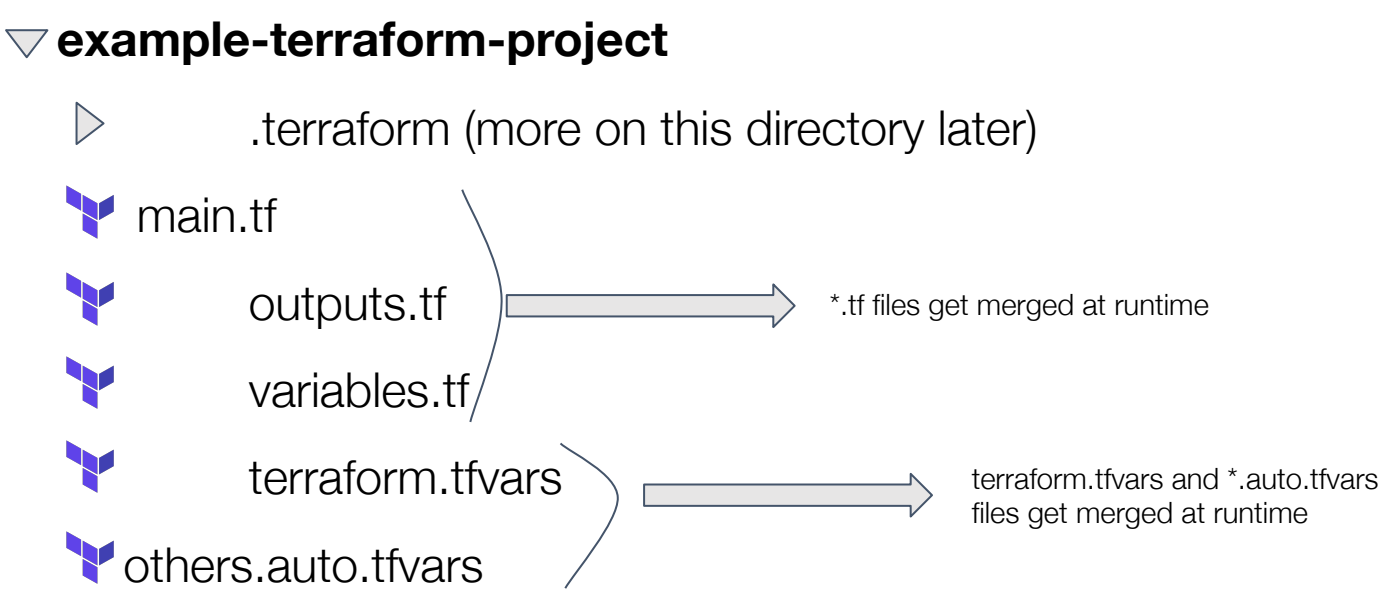
# Exercise Prep: Let's Get Set Up

1. You will be given now username and password for a sandbox environment in AWS

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

3. Access the repo of the course (if you haven't already) and go to exercises/0-setup

# Format of a Terraform Project



# 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

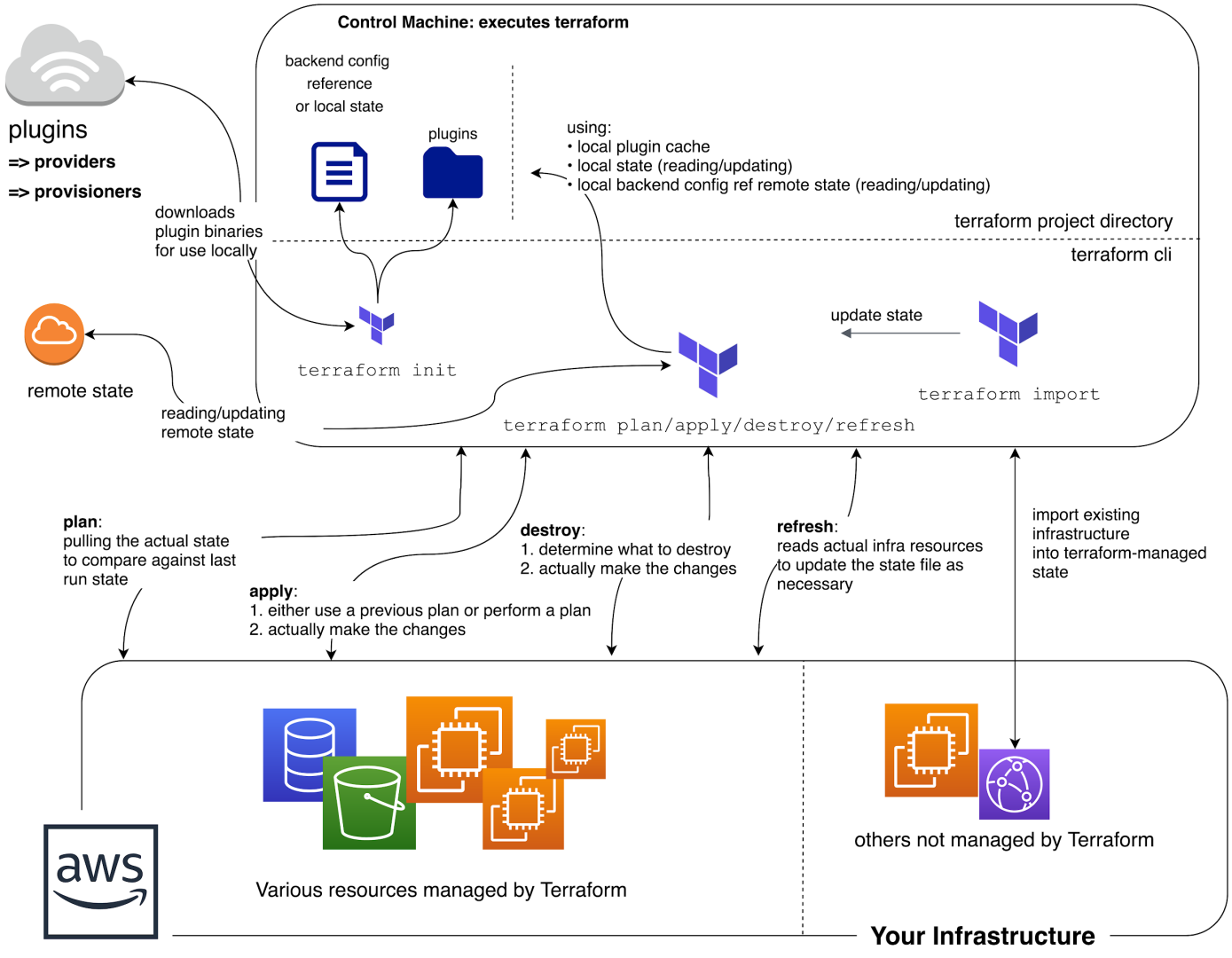
# Terraform Commands and the CLI

- 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`**)


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)
plugins
=> providers
=> provisioners
downloads
plugin binaries
for use locally
terraform project directory
terraform cli
remote state
terraform init
update state
terraform import
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`

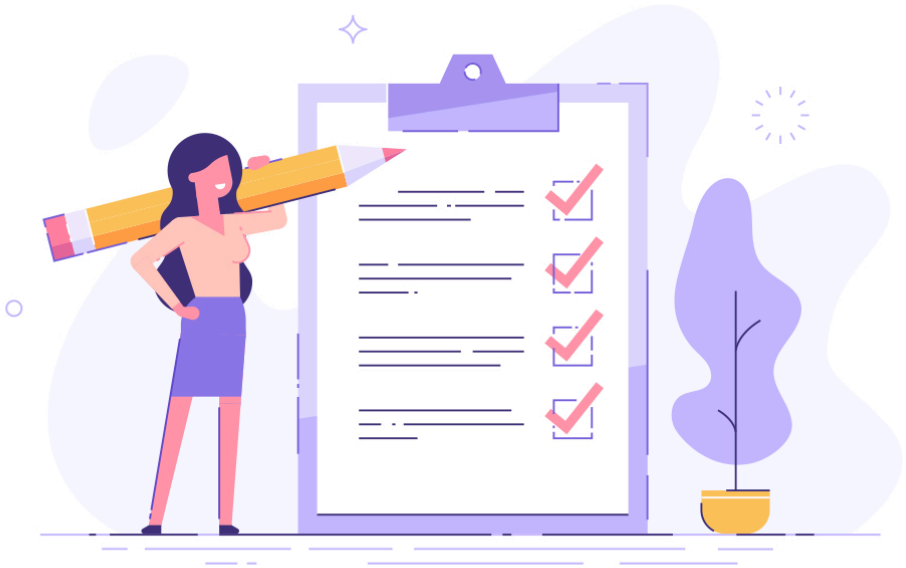
- a special command, run before other commands/operations
- what does it do?
  - downloads required provider packages
  - downloads modules referenced in the HCL (more on modules later)
  - initializes state
    - local state: ensuring local state file(s) exist
    - remote state: more complex initialization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

# LAB: First project

- Instantiate a provider
- Create a resource



# Extending Your Project

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

# Input Variables

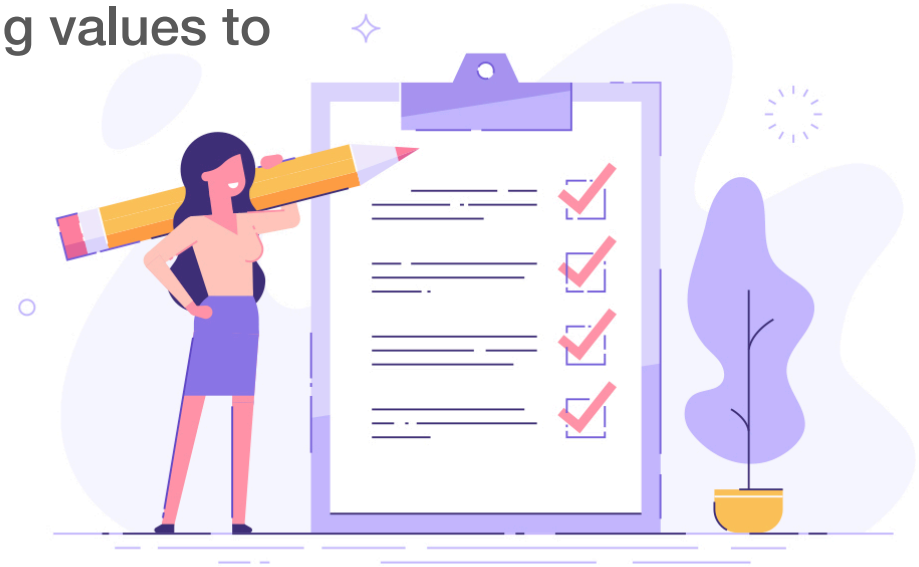
- enable interchangeable values to be stored centrally and referenced single or multiple times
- similar to variables in other languages
- declared in **`variable`** stanzas
- parsed first
- cannot interpolate or reference other variables
- allow for default values
- optionally specify value type, e.g.,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

# Example Variable Definition

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

# LAB 2: Variables

- Understand the ways of assigning values to variables



# 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: AMI lookup

```
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"]
  }
}
```

# Discussion

- Check examples/2-data-sources
- Is it clear? Let’s chat about what the configuration does!

# 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
- Used for post-config, but generally bad idea (except some use cases)
- 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: `null_resource`

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

# Provisioner Example: `remote-exec`

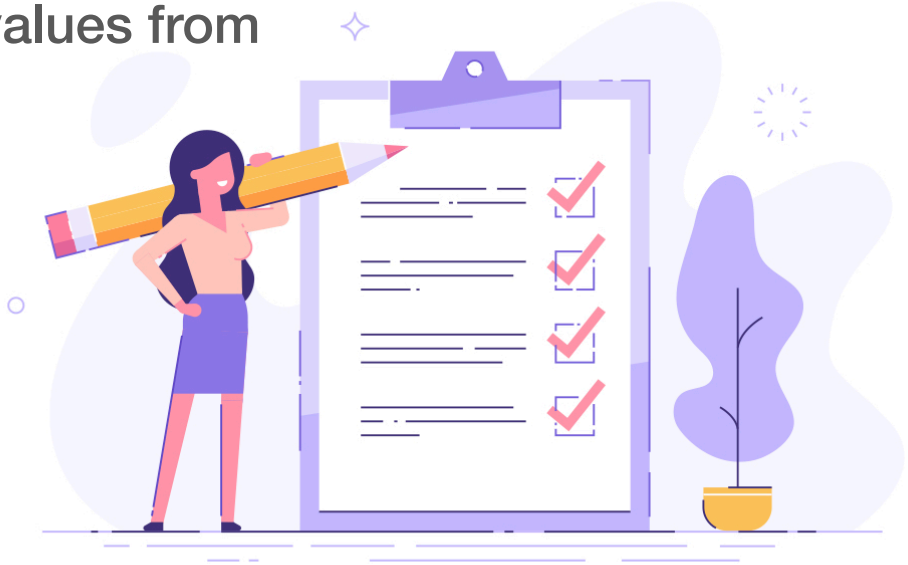
```
resource “aws_instance” “web” {
  ...

  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"
              ]
  }
}
```

# Discussion

- Check examples/3-provisioners
- Is it clear? Let’s chat about what the configuration does!

# LAB 3: Data Sources

- Understand the ways of getting values from resources



# Terraform Internals

# 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)

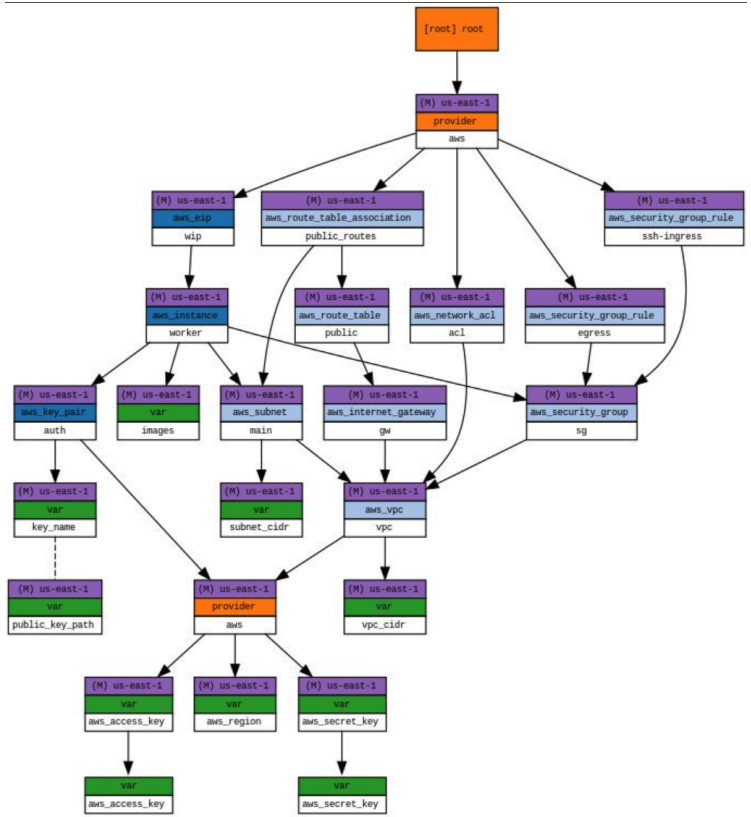
- examples
  - S3
  - swift
  - http
  - Terraform Enterprise
  - etc.

# How to Query or See the Current State

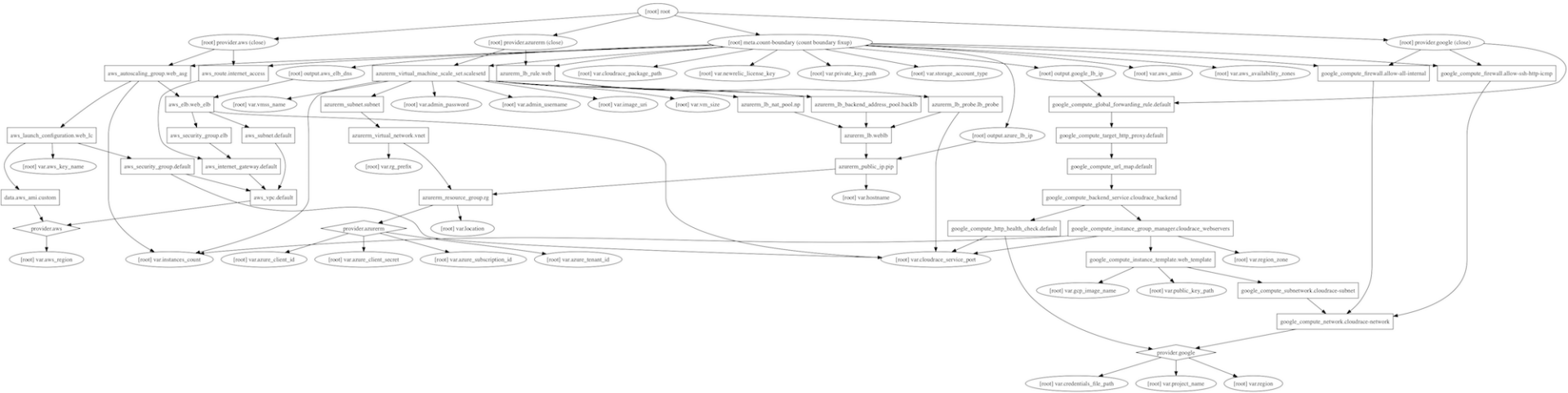
- CLI
  - **`terraform show [-json]`**
    https://www.terraform.io/docs/commands/show.html

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

# Visualizing Graph Outputs



# Visualizing Graph Outputs



# Executing Plans: `terraform apply`

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

# LAB 4: Plans and Applies

- Understand the ways of assigning values to variables

# 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]”
}
```

# Reusability Patterns

# 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

# Outputs

- *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 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

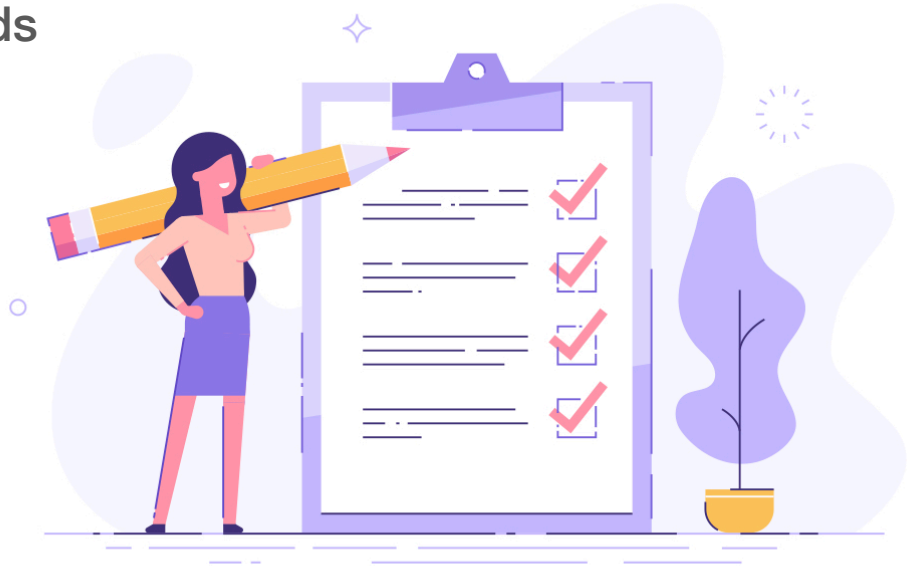
```
module “my_module” “printer” {
  source = “./my_module”

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

# Discussion

- Check examples/4-modules
- Is it clear? Let’s chat about what the configuration does!

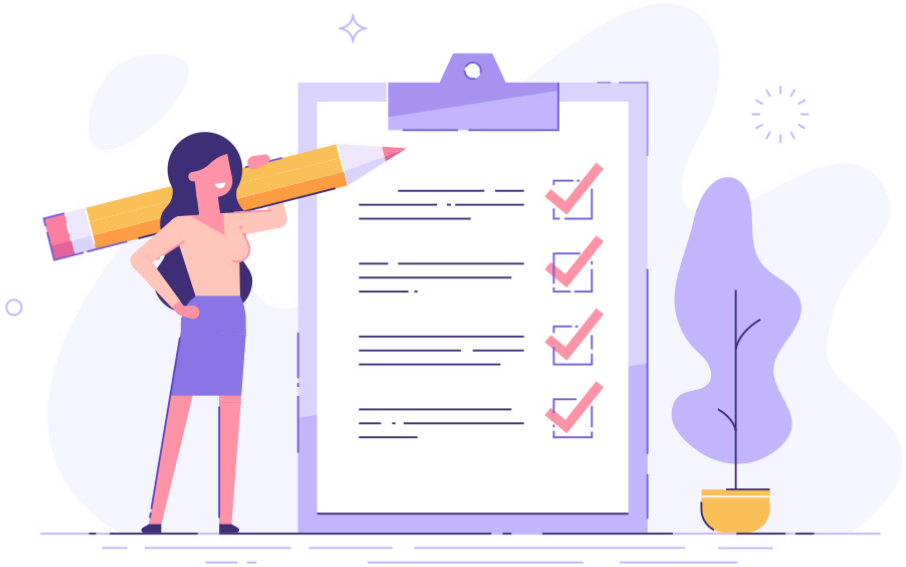
# LAB 5: Querying local state

- Use data sources with other backends



# LAB 6: Querying remote state

- Configure remote backends
- Query remote states

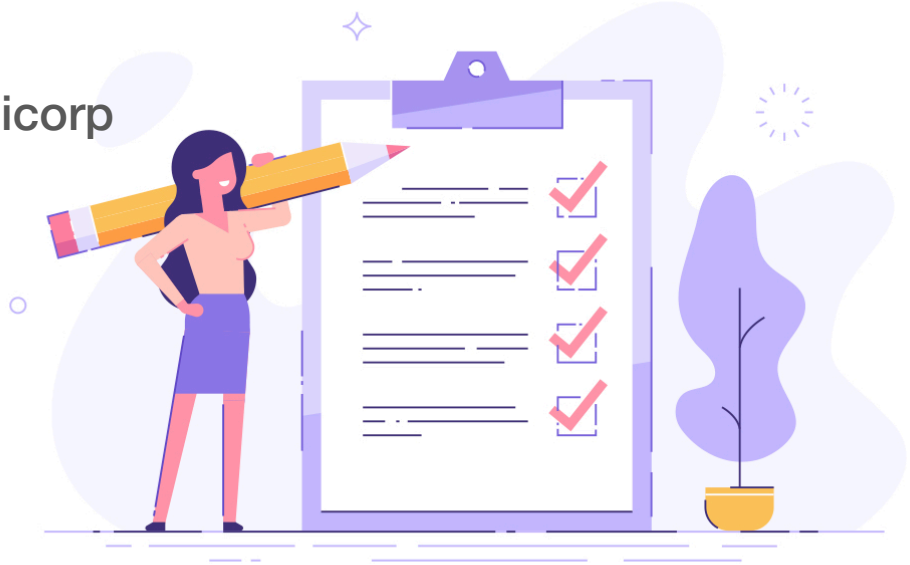


# LAB 7: Interacting with Providers

- Create multiple instances of a provider
- Perform multi-client_id deployments

# LAB 8: Modules

- Use local modules
- Use remote modules signed by Hashicorp

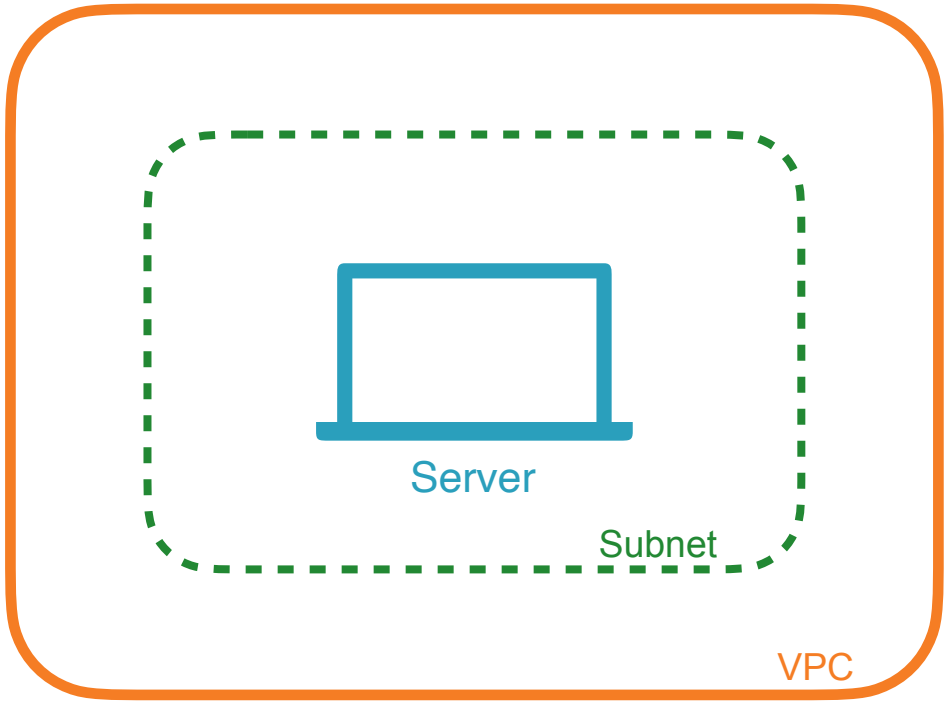


# Experimentation / Hack time

# Experimentation Time: Webserver

- Let's first do this by hand, then use Terraform
- Create an Ubuntu instance, type t2.micro, provision this instance to install **nginx**
  - make sure we can access **nginx** from the outside world
  - use online docs for help, use each other, and I'm here too if you need
  - Eventually you should have a public IP or DNS record such that accessing through the browser you get the Nginx traditional index.html page

# Architecture



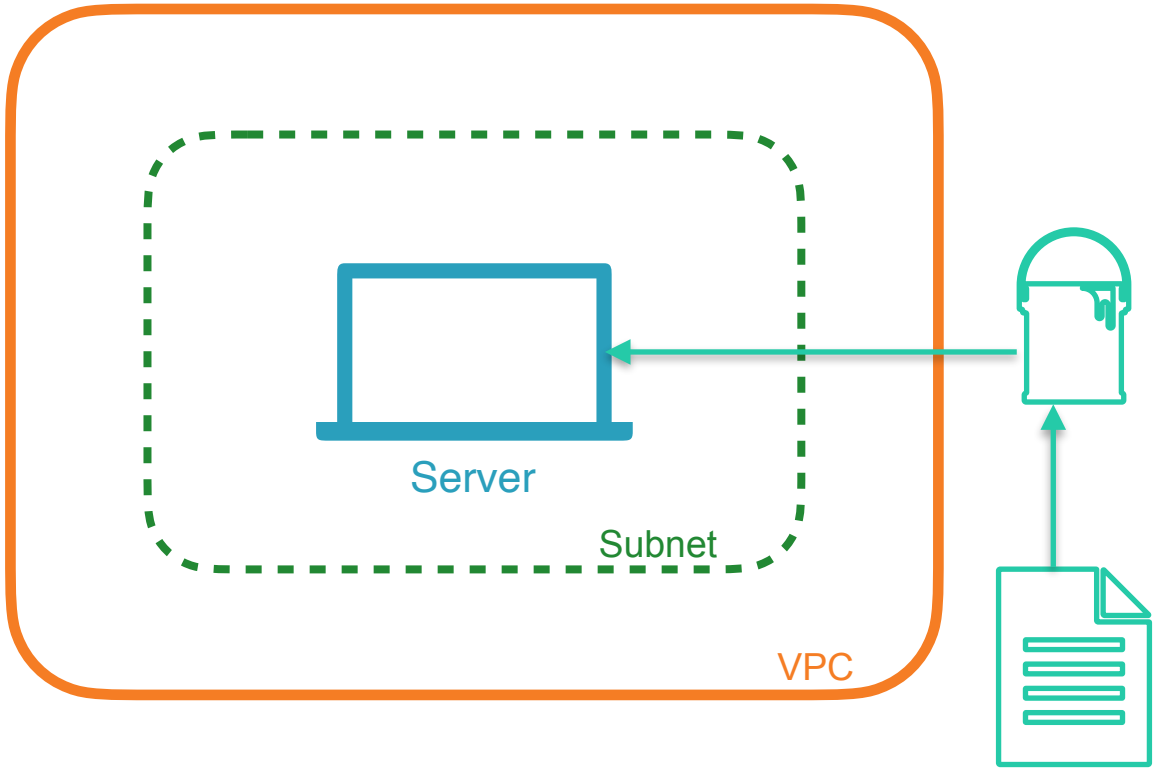
# Experimentation Time: Webserver (cont'd)

- You can go in many different directions as you solve this, and there are plenty of things to play with along the way and after, some ideas:
  - imagine your Terraform code to spin up this instance needed to be reused, what are your options–make it reusable!
    - **hints**: package as a module
  - pretend that some other automation needed the public IP from the created instance, how would you do this?
    - **hints**: outputs, and play with wrapping Terraform commands in a parent script (e.g., bash or Python)
  - ...and finally, ANYTHING else you'd like to start learning more about, **`terraform workspaces`** maybe?

# Experimentation Time: Webserver (cont'd)

- If you want to get more complex, you can create a bucket, upload some html website, and make the nginx instance serve that content instead.
  - For that you would need an IAM instance profile and attach it to the instance, such that it has access to S3. If you finished with the rest ping me for the correct policies to do this.

# Architecture



# Congrats!

**Now you know enough to create resources with the docs. We will next learn to make your code less verbose, and use more expressions to avoid duplicating yourself!**

# Debugging Terraform

# 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., Azure rejecting a VM for insecure password)

# 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
  - Using -recursive it checks all dirs

# 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.,

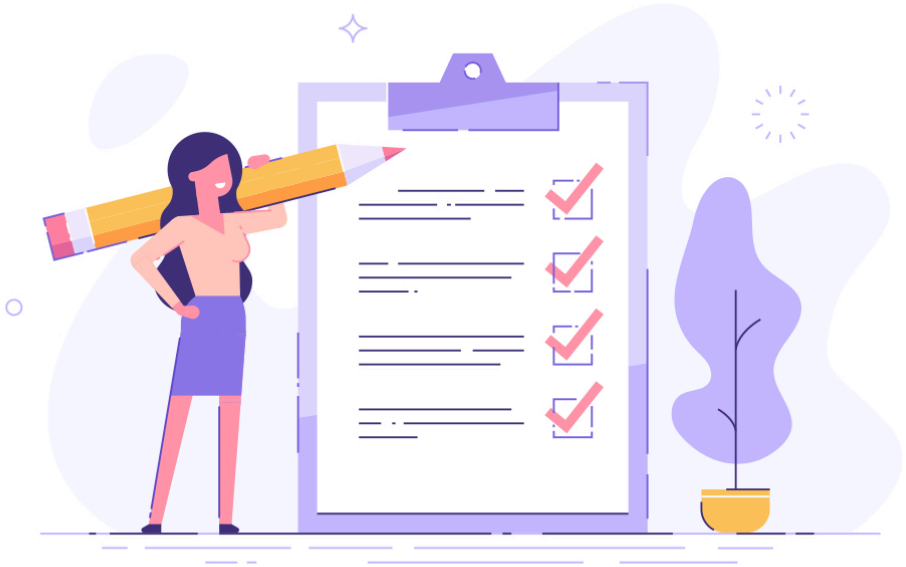
```
terraform graph | dot -Tsvg > graph.svg
```

# Troubleshooting: `terraform console`

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

# LAB 9: Error Handling

- Handling errors



# Terraform
# Expressions and Functions

# 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://developer.hashicorp.com/terraform/language/expressions/strings

# 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://developer.hashicorp.com/terraform/language/functions

# 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.images["us-east-1"]}`** would get the value of the **`us-east-1`** key within the **`images`** 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 Meta-argument

- 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

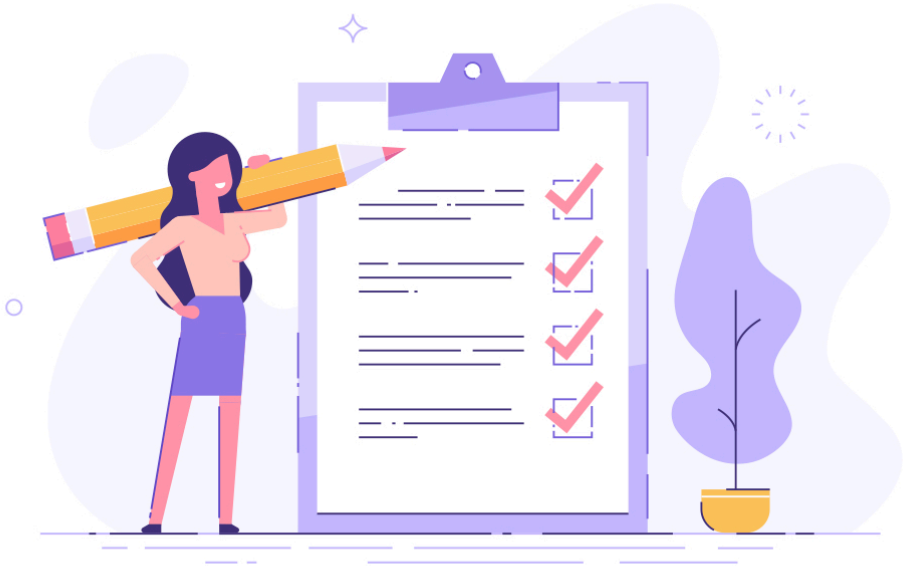
```
resource “aws_instance” “web” {
  ami           = “${var.ami}“
  instance_type = “${var.instance_type}”

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

# Discussion

- Check examples/5-expressions
- Is it clear? Let’s chat about what the configuration does!

# LAB 10: Understanding & Manipulating Data/Variables

- Manipulate primitive types
- Manipulate complex types
- Use expressions



# 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

- **Splat syntax**

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

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

```
${aws_instance.web.*.id}
```

# Terraform Expressions

- **`for`** expressions to convert lists/maps/tuples/objects to other lists/maps/tubles/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 (new in v0.12)

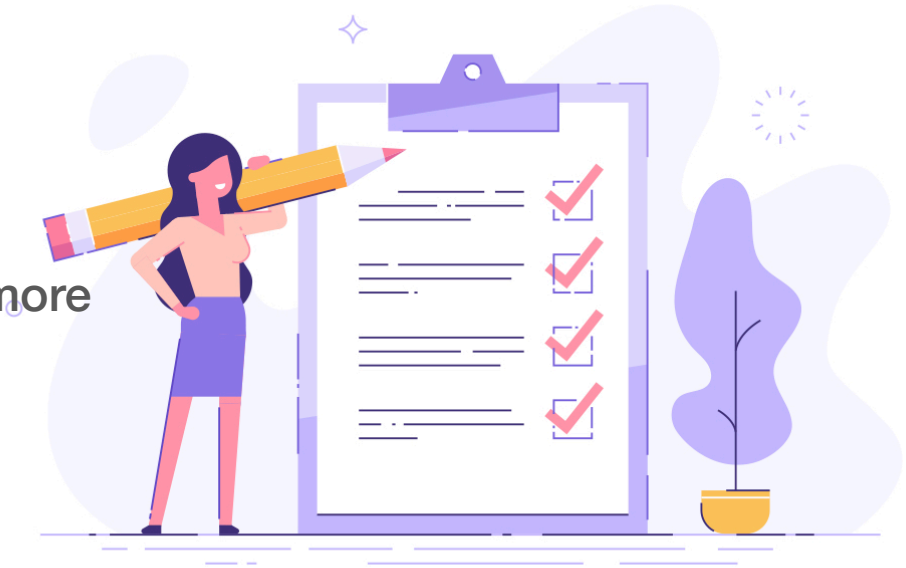
```
resource "aws_security_group" "example" {
  name = "example" # can use expressions here

  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

# LAB 11: Expressions and Meta-Arguments

- Create multiple resources with count
- Create multiple resources with for expression
- Use built in functions to make code more dynamic



# Handling drift

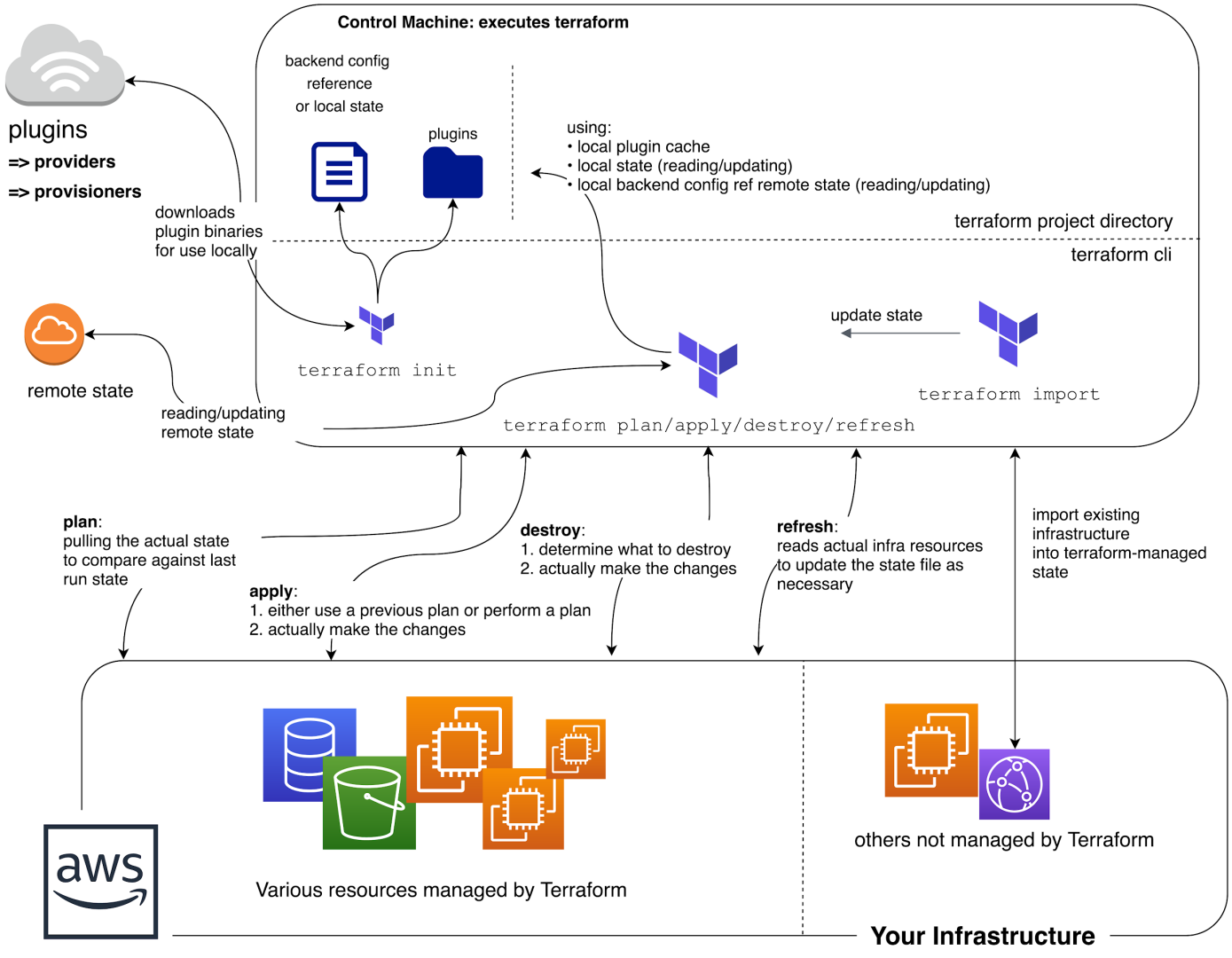
# Keeping Terraform in Sync with Infra

- 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

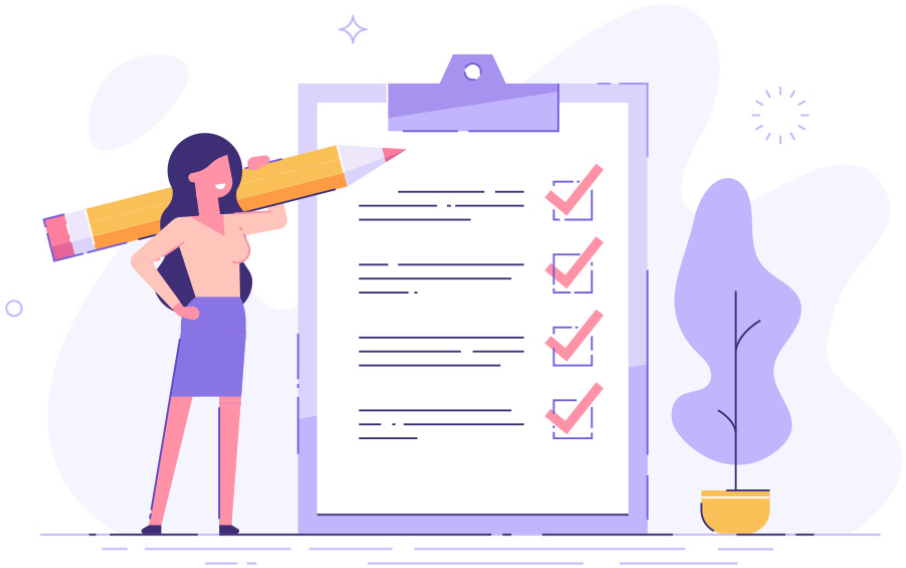
- what if we want to keep the changes?
- you can import them
  - use **`terraform import`** to pull in the changes to the state
  - must also change the Terraform config to match any changes
  - if you have a clean plan with no planned changes, you were successful
- e.g.,

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


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)
plugins
=> providers
=> provisioners
downloads
plugin binaries
for use locally
terraform project directory
terraform cli
remote state
terraform init
update state
terraform import
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

# LAB 12: Handling configuration drift

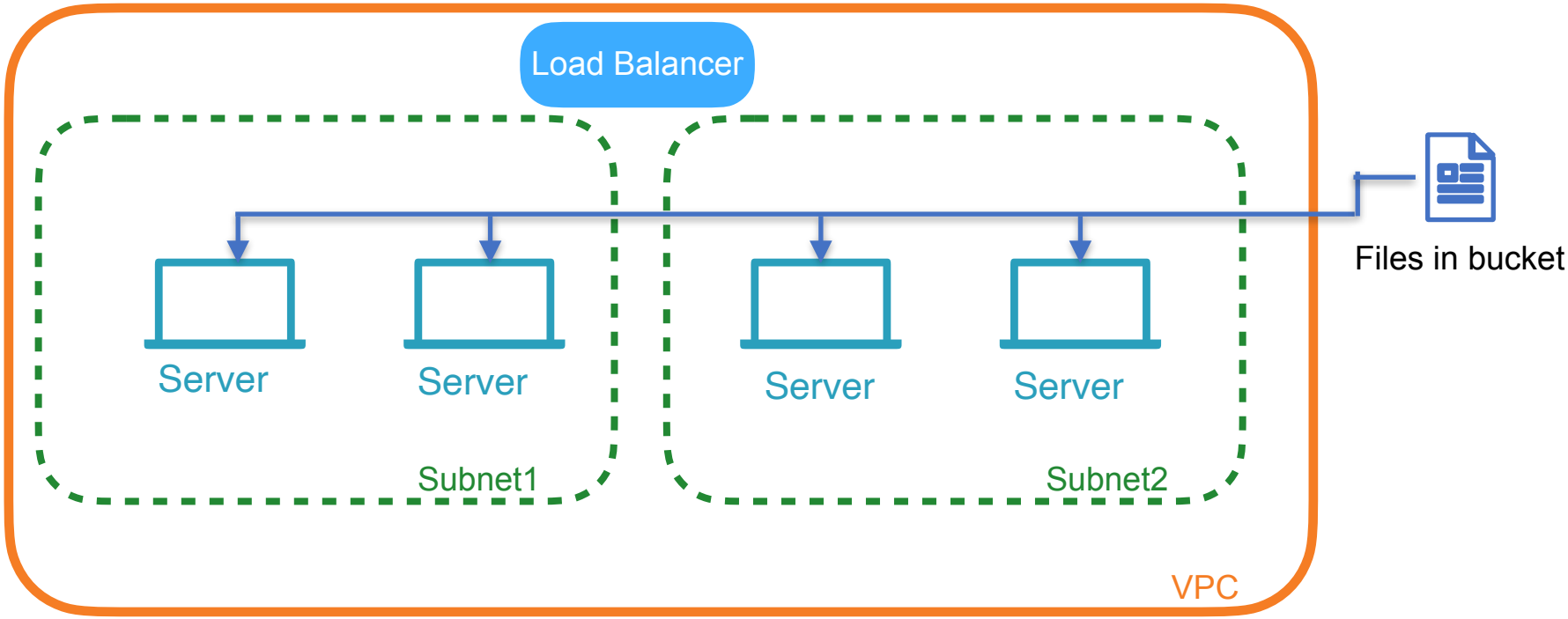
- Handle drift by reversing
- Handle drift by importing



# Experimentation / Hack time

# Hack time: Production Webserver

- Our previous web server was OK for a development configuration, but now let's go real to prod. We need to define a production environment with:
  - 4 copies of our web server in 2 different subnets in 2 different availability zones
  - 1 load balancer to balance traffic among the 4 instances
  - We should be able to ssh into any instance
  - Pro version: Add a S3 bucket container, add a custom website file to it, and make each instance to fetch the index.html from that bucket. Don't forget about the IAM role for that


Load Balancer
Server
Server
Server
Server
Subnet1
Subnet2
VPC
Files in bucket

***YOU'RE NOW READY FOR THE WILD WORLD OF TERRAFORM***

*THANKS FOR BEING HERE!*