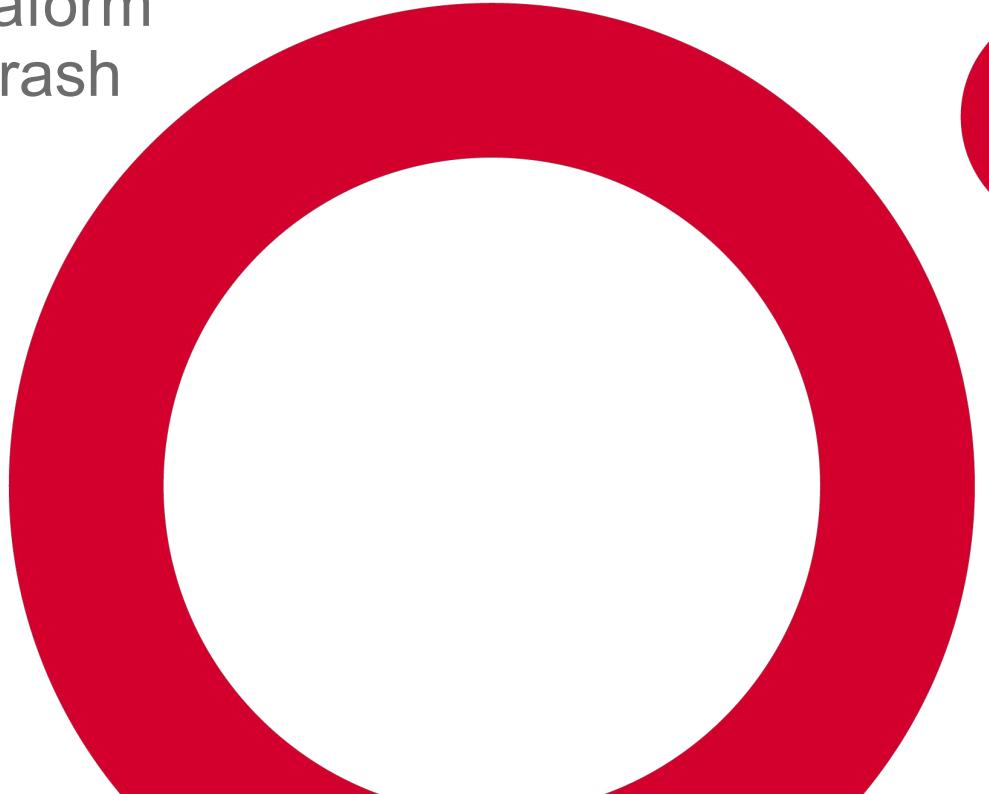




Hashicorp Certified: Terraform Associate Certification Crash Course

Terraform 1.3.3 Edition



About the trainer



bmuschko



bmuschko



bmuschko.com



 **AUTOMATED
ASCENT**
automatedascent.com

Certification Exam

Objectives, Curriculum, Prerequisites, Learning
Resources

Exam Objectives

“Basic understanding of concepts and skills associated with open source HashiCorp Terraform and Cloud features.”



The certification program allows users to demonstrate their competence in a multiple choice test.

<https://www.hashicorp.com/certification/terraform-associate>



The Curriculum

1	Understand infrastructure as code (IaC) concepts
1a	Explain what IaC is
1b	Describe advantages of IaC patterns

2	Understand the purpose of Terraform (vs other IaC)
2a	Explain multi-cloud and provider-agnostic benefits
2b	Explain the benefits of state

3	Understand Terraform basics
3a	Install and version Terraform providers
3b	Describe plugin-based architecture
3c	Write Terraform configuration using multiple providers
3d	Describe how Terraform finds and fetches providers

4	Use Terraform outside of core workflow
4a	Describe when to use <code>terraform import</code> to import existing infrastructure into your Terraform state
4b	Use <code>terraform state</code> to view Terraform state
4c	Describe when to enable verbose logging and what the outcome/value is

5	Interact with Terraform modules
5a	Contrast and use different module source options including the public Terraform Module Registry
5b	Interact with module inputs and outputs
5c	Describe variable scope within modules/child modules
5d	Set module version



The Curriculum

6	Use the core Terraform workflow
6a	Describe Terraform workflow (Write -> Plan -> Create)
6b	Initialize a Terraform working directory (<code>terraform init</code>)
6c	Validate a Terraform configuration (<code>terraform validate</code>)
6d	Generate and review an execution plan for Terraform (<code>terraform plan</code>)
6e	Execute changes to infrastructure with Terraform (<code>terraform apply</code>)
6f	Destroy Terraform managed infrastructure (<code>terraform destroy</code>)
6g	Apply formatting and style adjustments to a configuration (<code>terraform fmt</code>)

8	Read, generate, and modify configuration
8a	Demonstrate use of variables and outputs
8b	Describe secure secret injection best practice
8c	Understand the use of collection and structural types
8d	Create and differentiate resource and data configuration
8e	Use resource addressing and resource parameters to connect resources together
8f	Use HCL and Terraform functions to write configuration
8g	Describe built-in dependency management (order of execution based)

7	Implement and maintain state
7a	Describe default local backend
7b	Describe state locking
7c	Handle backend and cloud integration authentication methods
7d	Differentiate remote state back end options
7e	Manage resource drift and Terraform state
7f	Describe backend block and cloud integration in configuration
7g	Understand secret management in state files

9	Understand Terraform Cloud capabilities
9a	Explain how Terraform Cloud helps to manage infrastructure
9b	Describe how Terraform Cloud enables collaboration and governance



Candidate Prerequisites



Basic Terminal Skills



Basic understanding of on premises and cloud architecture



Preparing for the Exam

Documentation is not allowed during the test

- Study Guide:

<https://learn.hashicorp.com/tutorials/terraform/associate-study>

- Exam Review:

<https://learn.hashicorp.com/tutorials/terraform/associate-review>

- Sample Questions:

<https://learn.hashicorp.com/tutorials/terraform/associate-questions>



Infrastructure as Code (IaC)

Concepts and Benefits

Infrastructure as Code (IaC)



Manage infrastructure with the help of code

- Treats all aspects of operations as software via configuration
- Code is tracked in a SCM repository
- Automation makes the provisioning process consistent, repeatable, and updates fast & reliable



Repeatable Process



Clear instructions that describe the desired state

- A set of instructions are defined with the help of a declarative language
- Operations are idempotent e.g. an update to the environment will only make necessary changes but not duplicate what already exists
- **Example:** A database was deployed with a port but the port needs to change. We can simply change the port number in the code we used to deploy the database in the first place.



Consistent Environments



Environment should look extremely similar

- Projects often use a variety of deployment environments e.g. development, staging, and production
- The same automation code can be used to provision infrastructure so that it looks consistent across all environments
- **Example:** Deploying an EC2 server across all environments but with slightly different regions



Reusable Functionality



Configuration can be abstracted and applied to a set of projects

- Configuration is defined with the help of code
- Code can be shared across different repositories
- **Example:** Deploying a HTTP web server for multiple projects with the same piece of code.



Self-Documenting



Source code represents the architecture

- Each piece of infrastructure has been described with a set of instructions
- No more guesswork on what configuration has been used to provision infrastructure
- **Example:** New engineering team members can read source code to understand how infrastructure is configured



Financial Savings



Increased efficiency, less mistakes through automation

- Reduced risk due to minimizing human error
- Infrastructure can be verified with automated tests
- IaC functions can be used to spin down environments during times of less traffic
- **Example:** Decrease the manual grunt work for DevOps personnel and spending it on mission-critical tasks instead



Q & A



Terraform 101

What is Terraform and How Does it Implement IaC?

What is Terraform?



Infrastructure automation tool

- Open source and cloud provider-agnostic
- Configuration is expressed in a declarative language with either HashiCorp Configuration Language (HCL) or JSON
- Deployment of infrastructure happens with a push-based approach (no agent to be installed on remote machines)



How Does Terraform Work?



Binary makes API calls to cloud providers

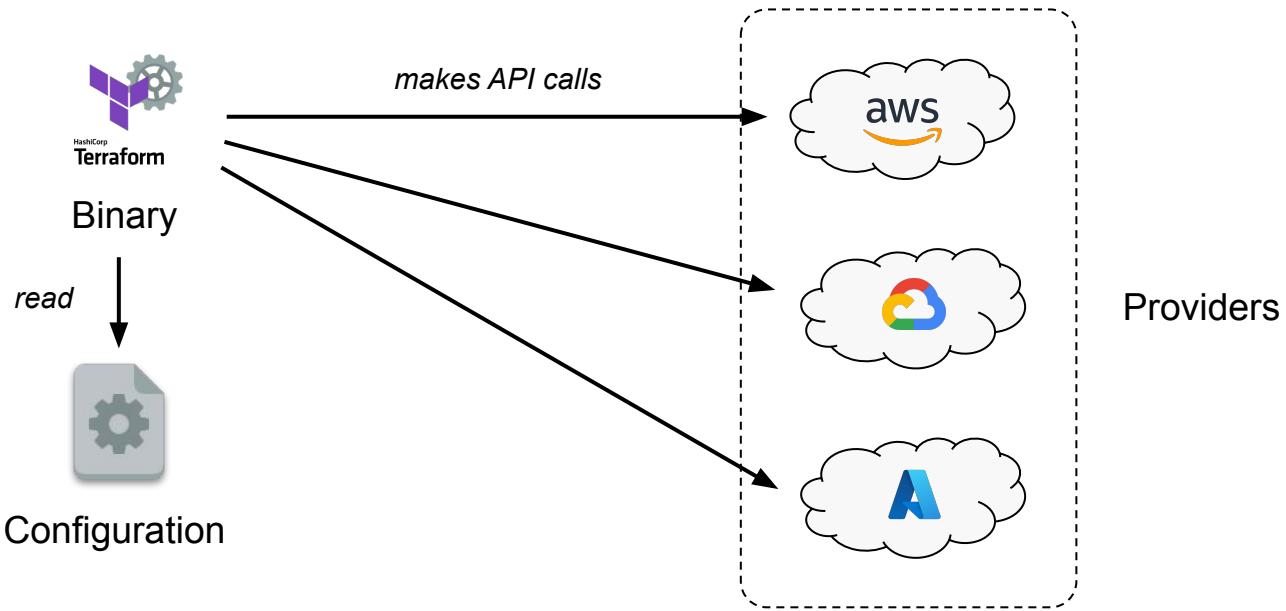
- CLI tool for deploying infrastructure to one or many cloud provider(s)
e.g. AWS, Azure, or Google Cloud
- Under the hood, makes API calls on behalf of a *provider* including authentication mechanisms
- Terraform *configurations* are the codified instructions in the form of a text file that tell it which API calls to make



Interaction with Cloud Providers



Binary makes API calls to cloud providers



Cloud Provider Portability



Features are different, Terraform's technical approach is not

- Cloud providers don't support the exact same infrastructure
- Terraform allows you to use the same approach to define provider-specific configuration
- You use the same Terraform language, toolset, and IaC practices



Terraform Components



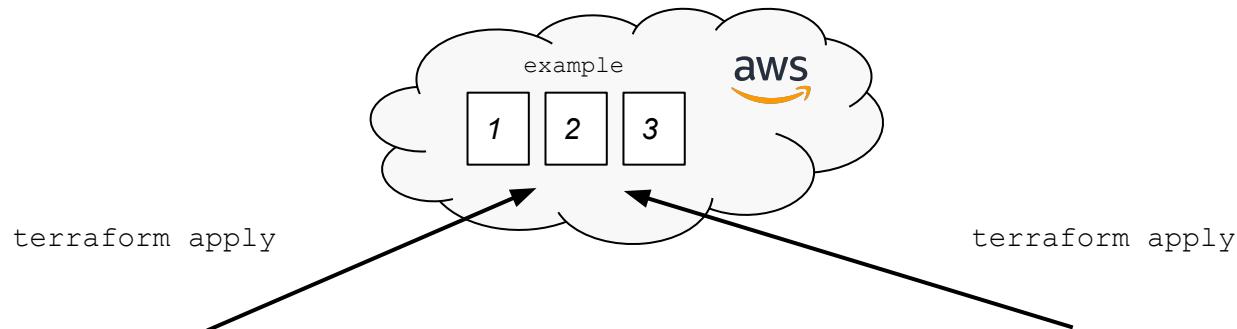
Key building blocks in architecture

- *Executable*: Binary run from the command line that contains Terraform's core functionality
- *Configuration file(s)*: Files with the extension `.tf` or `.tfvars` that define the desired configuration for provisioning infrastructure
- *Provider plugins*: Executables invoked by Terraform to interact with cloud provider APIs, hosted on a registry
- *State data*: The desired configuration and its current state



Persisting State

Knows what has been created before and applies changes



```
resource "aws_instance" "example" {  
  count      = 2  
  ami        = "ami-0c55b159cbfafef0"  
  instance_type = "t2.micro"  
}
```



```
resource "aws_instance" "example" {  
  count      = 3  
  ami        = "ami-0c55b159cbfafef0"  
  instance_type = "t2.micro"  
}
```

State Storage

Where and how is state stored and updated?

- Stores state in internal database, each resource is represented by a key-value pair in the entry
- Any changes to the resources will be reflected in the state
- Locally, the state is saved in the file `terraform.tfstate`
- Remote state handling exists to support consistent team collaboration



Benefits of State



Keeping state fulfills important requirements

- *Dependencies:* Resources can have dependencies on each other, Terraform retains this metadata to be able to safely perform operations e.g. delete
- *Performance:* Terraform stores a cache of the attribute values for all resources in the state for performance reasons
- *Consistency:* Terraform employs locking to avoid synchronization and collaboration issues





Terraform and Ansible

Differences and use cases

- **Ansible:** Configuration management tool for installing/configuring software and tools in the infrastructure
- **Terraform:** May invoke Ansible after infrastructure has been deployed
- When to involve Ansible?
 - You already have Ansible playbooks you want to invoke
 - Nowadays, it's more common to create an immutable image that already contains the software needed



Terraform and Packer

Differences and use cases

- **Packer:** Creates machine images for multiple platforms
- **Terraform:** Uses machine image to provision infrastructure
- When to involve Packer?
 - To shorten deployment time by baking needed functionality into AMI
 - To simplify or avoid logic in Terraform needed for installing additional tooling, monitoring, logging etc. on top of base AMI



Terraform and Consul



Do I need to involve a state backend?

- **Terraform** can store state data in a backend to improve team collaboration
- **Consul** is one of the configurable backends supported by Terraform. Effectively, it's a key-value store installed on a dedicated server
- When to involve Consul?
 - When your team works on Terraform configuration in parallel and runs into potential conflicts a lot
 - If you do not want to buy into Terraform Cloud



Q & A



BREAK



Terraform Installation and Configuration

Binary Installation, AWS Account, IDE Support

Installing Terraform



Easy to install on all operating systems

- Manual installation
 - Download ZIP file containing the pre-compiled binary
 - Add binary to PATH environment variable
- Using a package manager
 - Available via Homebrew and Chocolatey
 - Takes care of adding binary to terminal



Verifying and Using Terraform

The `terraform` executable is the main entry point

```
$ terraform version
Terraform v1.3.1
on darwin_amd64
```

```
$ terraform -help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

  ...


```



Switching Between Versions

tfswitch lets you manage and use versions in parallel



```
$ tfswitch 1.2.1
$ terraform version
Terraform v1.2.1
on darwin_amd64
```

```
$ tfswitch 1.3.2
$ terraform version
Terraform v1.3.2
on darwin_amd64
```





Setting Up an AWS Account

Create new account at <https://aws.amazon.com>

The screenshot shows the AWS homepage with a dark blue background. At the top right, there is a navigation bar with links for Contact Us, Support, English, My Account, and Sign In. A prominent orange button labeled "Create an AWS Account" is located in the top right corner of the header. This button is highlighted with a red rectangular box and has a black arrow pointing towards it from the left side. Below the header, the main heading "Start Building on AWS Today" is displayed in white. A descriptive paragraph follows, stating: "Whether you're looking for compute power, database storage, content delivery, or other functionality, AWS has the services to help you build sophisticated applications with increased flexibility, scalability and reliability." A "Get Started for Free" button is located below this text. The page is divided into two main sections: "For Builders" and "For Decision Makers". The "For Builders" section includes icons for launching an application and connecting with others, along with text about starting with step-by-step tutorials. The "For Decision Makers" section includes icons for optimizing business value and reinventing with data, along with text about reducing costs and improving productivity.





Setting Up an AWS Account

Free tier is sufficient for exercises in this course

aws

Sign up for AWS

Explore Free Tier products with a new AWS account.

To learn more, visit aws.amazon.com/free.

Root user email address
Used for account recovery and some administrative functions

AWS account name
Choose a name for your account. You can change this name in your account settings after you sign up.

Verify email address

OR

Sign in to an existing AWS account

Basic support - Free

- Recommended for new users just getting started with AWS
- 24x7 self-service access to AWS resources
- For account and billing issues only
- Access to Personal Health Dashboard & Trusted Advisor

Free Trials

Short-term free trial offers start from the date you activate a particular service



12 months free

Enjoy these offers for 12-months following your initial sign-up date to AWS



Always free

These free tier offers do not expire and are available to all AWS customers





Retrieving AWS Credentials

Create a new access key and store in safe place

The screenshot shows the AWS Management Console navigation bar. From left to right, it includes: a triangle icon, a user icon, 'N. Virginia ▾', 'bmuschko ▾', 'Account ID: 0314-0087-8386' with a copy icon, and a dark blue background area containing the following items:

- Account
- Organization
- Service Quotas
- Billing Dashboard
- Security credentials** (this item is highlighted with a red box)
- Settings

At the bottom right of this list is an orange 'Sign out' button.

The screenshot shows the 'Your Security Credentials' page. At the top, there's a header with three collapsed sections: 'Password', 'Multi-factor authentication (MFA)', and 'Access keys (access key ID and secret access key)' (the last one is expanded). Below the header, there's a note about using access keys for programmatic calls and a warning about sharing secret keys. A prominent blue button at the bottom left of the main content area is labeled 'Create New Access Key' and is also highlighted with a red box.

Created	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Create New Access Key						





Setting Provider Credentials

In current shell or from typical credentials location

```
$ export AWS_ACCESS_KEY_ID=<access-key-id>
$ export AWS_SECRET_ACCESS_KEY=<secret-access-key>
```

or

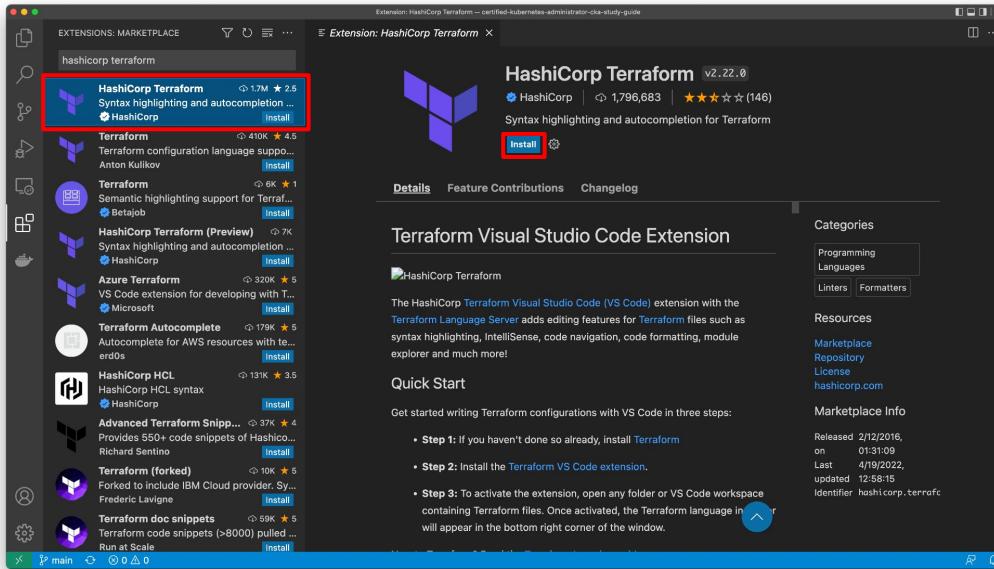
```
$ cat $HOME/.aws/credentials
[default]
aws_access_key_id=<access-key-id>
aws_secret_access_key=<secret-access-key>
```



IDE Integration with VSCode



Search for “HashiCorp Terraform” in the Extensions



EXERCISE

Installing Terraform
and Setting AWS
Credentials



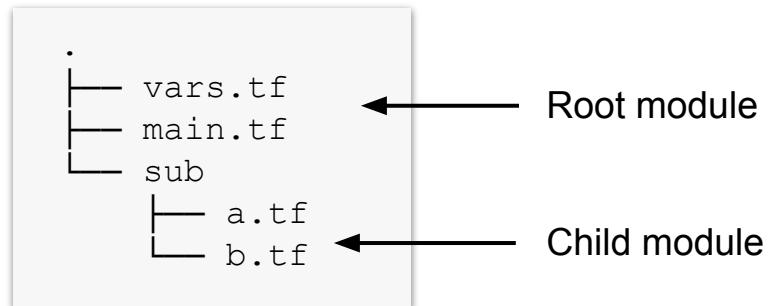
Getting Started With Terraform

Basic Concepts, Syntax, and Workflow

Files and Directories

Terraform configuration is defined by a collection of files

- *Configuration files* contain plain text instruction and have the extension `.tf` or `.tf.json`
- A *module* defines a set of (potentially versioned) configuration file(s)



Object Types

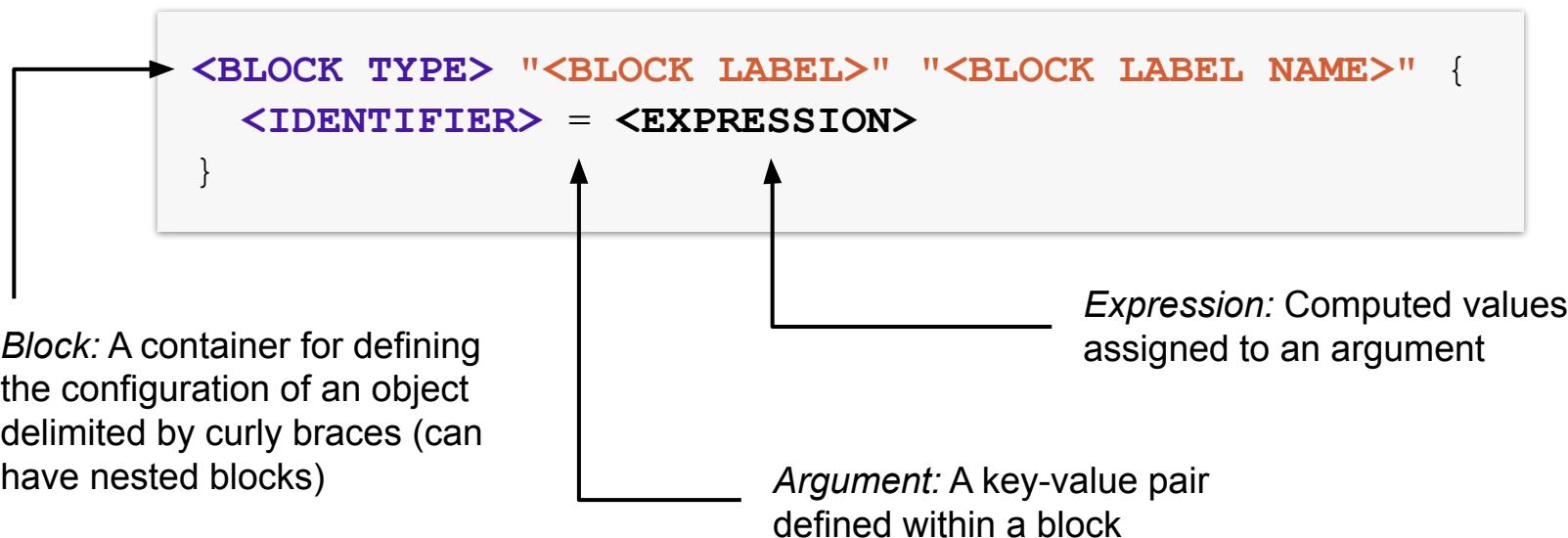
High-level elements used in a configuration file

- *Provider*: Allows Terraform to interact with a cloud provider through its API e.g. AWS or Azure, SaaS provider, or other APIs
- *Resource*: Defines the infrastructure pieces to be created in a target environment e.g. EC2 instance, a VPC, or a web server
- *Data source*: Can query information from a provider that can be used in the configuration e.g. a list of available AMIs



Configuration Syntax

HCL is preferred over JSON



Syntax Constraints

Just of a couple of little gotchas to look out for

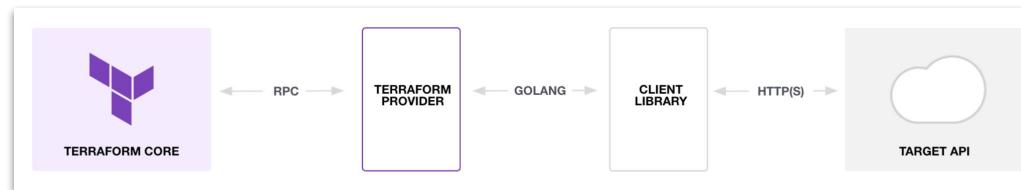
- *Identifiers* can contain letters, digits, underscores (_), and hyphens (-). The first character of an identifier must not be a digit, to avoid ambiguity with literal numbers.
- Comments can be defined by #, //, or /* */.
- Configuration files must always be UTF-8 encoded



Plugin-Based Architecture

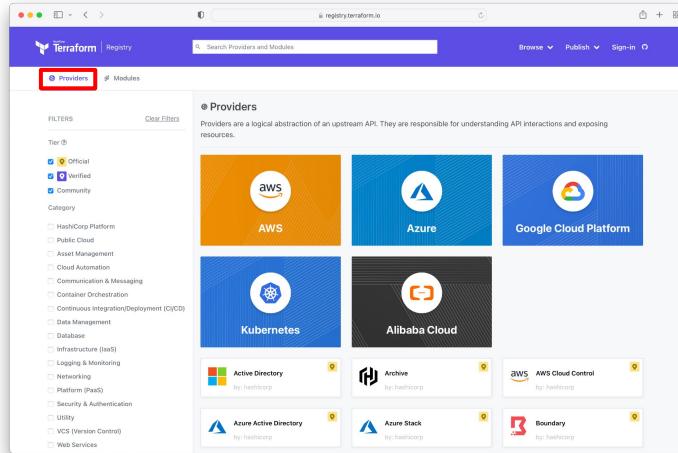
Core binary is small, providers are developed as plugins

- *Terraform Core:* The binary that communicates with plugins to manage infrastructure resources
- *Terraform plugins:* Executable binaries written in Go that communicate with Terraform core over an RPC interface



Exploring the Provider Registry

Central location for provider plugins and their documentation



<https://registry.terraform.io/browse/providers>



Defining Required Provider(s)

Sets up the properties for all providers of the assigned name

The free-form
name of the
provider

```
terraform {  
  required_providers {  
    aws = {  
      source  = "hashicorp/aws"  
      version = "4.17.1"  
    }  
  }  
}
```

The location of the
provider plugin in the
registry in the format
[<HOSTNAME>/] <NA
MSPACE>/<TYPE>

The version (range)
selector for the plugin



Version Constraint & Selection

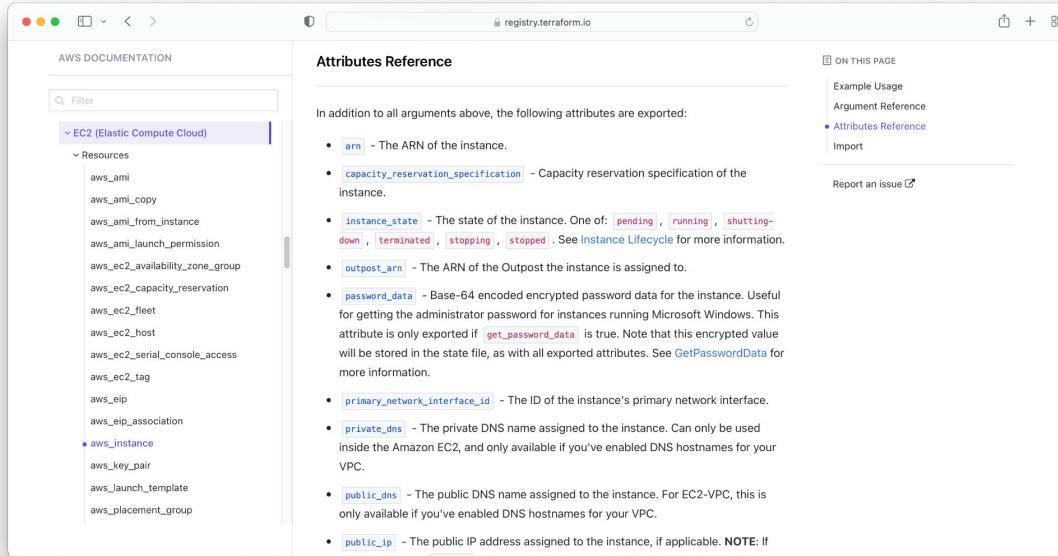
You can define a concrete version or a version range

Version Value	Meaning
no assigned value attribute	Picks the latest version of the provider available in registry
<code>>= 1.2.1</code>	Greater than or equal to the version 1.2.1
<code><= 1.2.1</code>	Less than or equal to the version 1.2.1
<code>~> 1.2.1</code>	Any version in the 1.2.x range up to the next minor version (1.3.0)
<code>>= 1.2.1, <= 1.5.0</code>	Any version between 1.2.1 and 1.5.0



Provider Documentation

A provider publishes the API (e.g. available attributes) on registry



Configuring Provider(s)

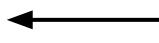
Need to be defined in the root module configuration file

```
provider "aws" {  
  region = "us-east-2"  
}
```



Default provider: Any resource that does not assign a provider explicitly will use this provider

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



Aliased provider: A resource can refer to this provider explicitly by assigning the provider argument (in this case aws.west)



Local Plugin Cache

Core binary is small, providers are developed as plugins

- Terraform automatically fetches plugins from the registry and stores them in the `.terraform` subdirectory
- Caching is enabled by default and a plugin version already available in the `.terraform` subdirectory will be reused
- Cache directory be configured with the environment variable `TF_PLUGIN_DIR` or the CLI option `--plugin-dir`



Configuring Provider Credentials

Provider exposes attributes for consuming credentials

```
provider "aws" {
  region = "us-east-2"
  access_key = "my-access-key"
  secret_key = "my-secret-key"
}
```



Do you not use this way to provide credentials!
The state file with store credentials in plain text.

```
provider "aws" {
  region = "us-east-2"
  shared_credentials_files = ["path-to-credentials-file"]
  profile = "profile-name"
}
```



Defining Resource(s)

Virtual server in AWS aka EC2 instance

```
resource "aws_instance" "web_server" {  
    ami              = "ami-0c55b159cbfafef0"  
    instance_type   = "t2.micro"  
}
```

The free or paid Amazon Machine Image (AMI) to run on EC2 instance available via the [AWS Marketplace](#)

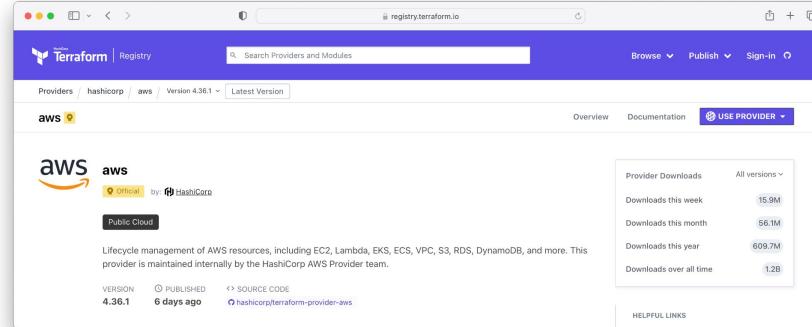
Type of EC2 instance which provides a certain amount of CPU, memory, disk space, and network capabilities



Provider is Derived from Resource

Defining a provider is optional, Terraform tries to anticipate

```
resource 'aws_instance' "web_server" {}
```



Use the prefix to determine provider on Terraform Registry and download latest version available



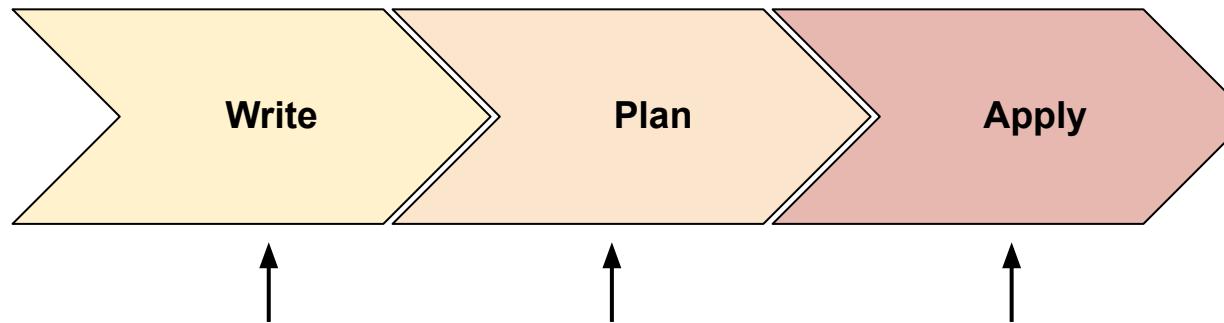
EXERCISE

Implementing a
Simple
Configuration File



Terraform Core Workflow

Three-step approach on a high-level

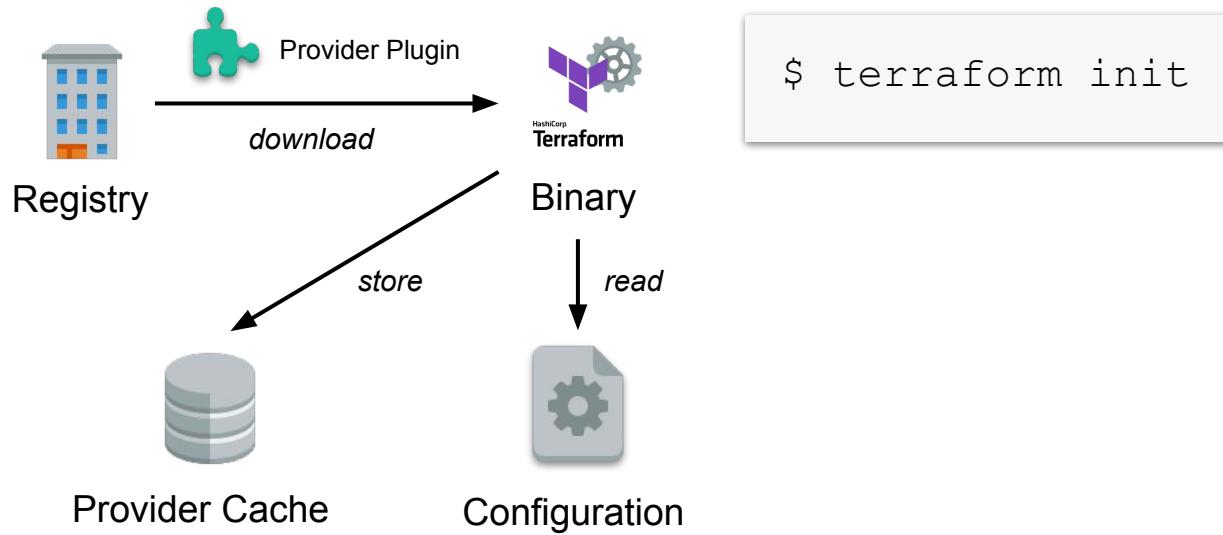


- Write configuration files and run `terraform init` to initialize the configuration to the target environment
- Source code and check it for syntax errors against the target environment
- Terraform syntax can be automatically formatted and validated



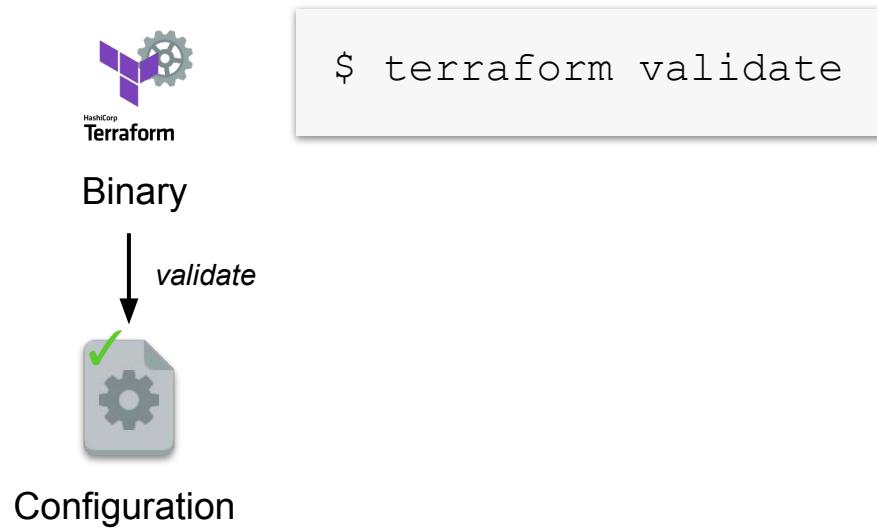
Initialize Working Directory

Prepare working directory, parse config, retrieve provider plugins



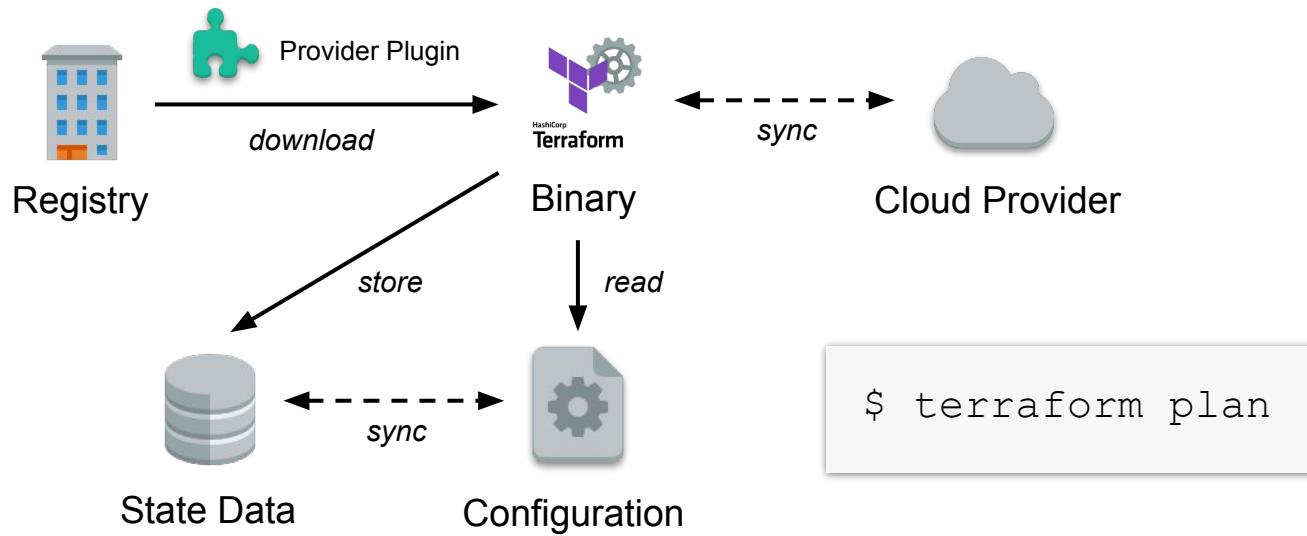
Validate Configuration

Check for syntax errors, doesn't guarantee successful deployment



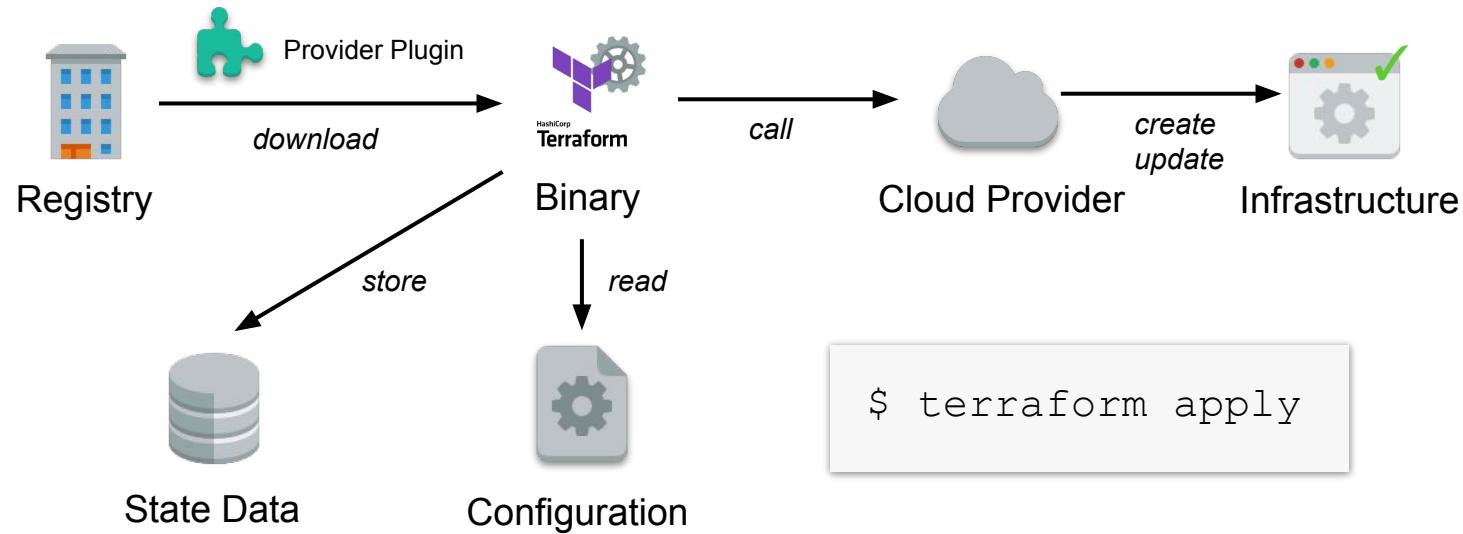
Plan Changes

Determine differences between local state and deployed infra



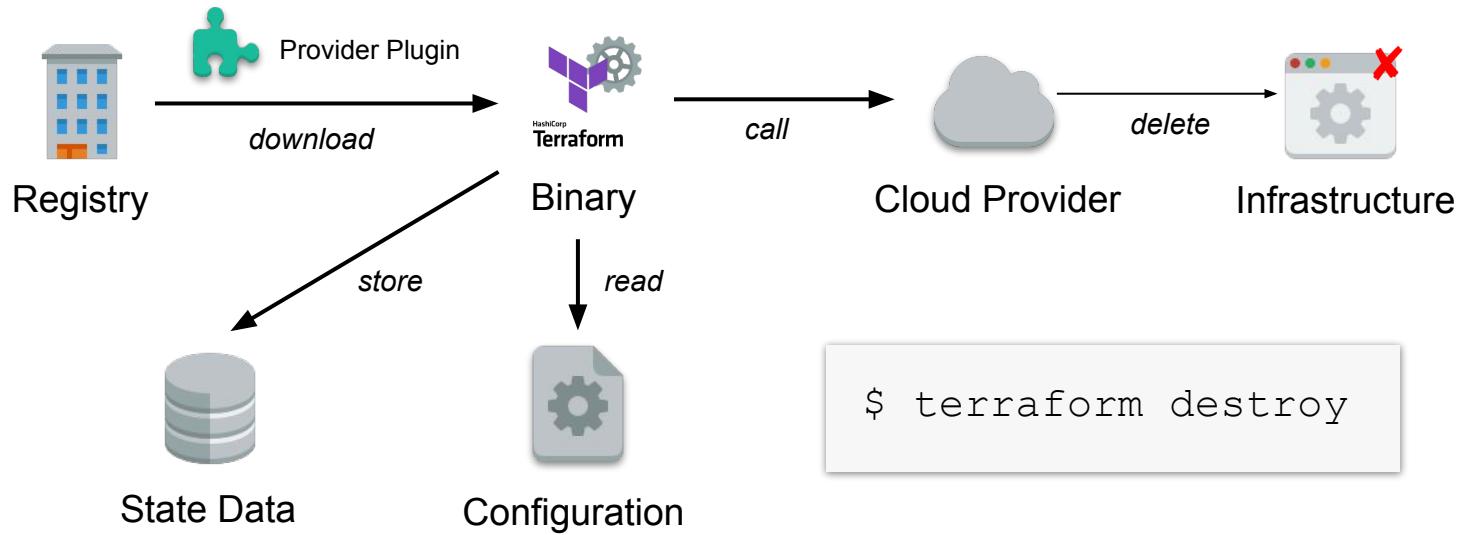
Apply Changes

Deploying the infrastructure or changing its delta



Destroy Infrastructure

Delete infrastructure in target environment based on state data



EXERCISE

Exercising the Core
Terraform Workflow



Local and Remote Provisioner

Using execution provisioners should be avoided (see [docs](#))

- After a resource has been created, you may want to run a script or operation locally (`local-exec`) or remotely (`remote-exec`)
- HashiCorp recommends to only use provisioners as the last resort as it cannot be guaranteed that a resource is in an operational state yet
- You can use infrastructure provisioning tools like Ansible and Chef or a custom image with the pre-installed tools instead



Local Provisioner

Invokes a local executable after resource creation (see [docs](#))

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```



Remote Provisioner

Invokes script/cmd on remote resource after creation (see [docs](#))

```
resource "aws_instance" "web" {
  # ...

  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh",
      "/tmp/script.sh args",
    ]
  }
}
```



Remote Execution on AWS

With AWS provider, user_data is similar to remote execution

```
resource "aws_instance" "web" {
  # ...

  user_data = <<-EOF
    "chmod +x /tmp/script.sh",
    "/tmp/script.sh args",
  EOF
}
```



Visible in AWS console



EXERCISE

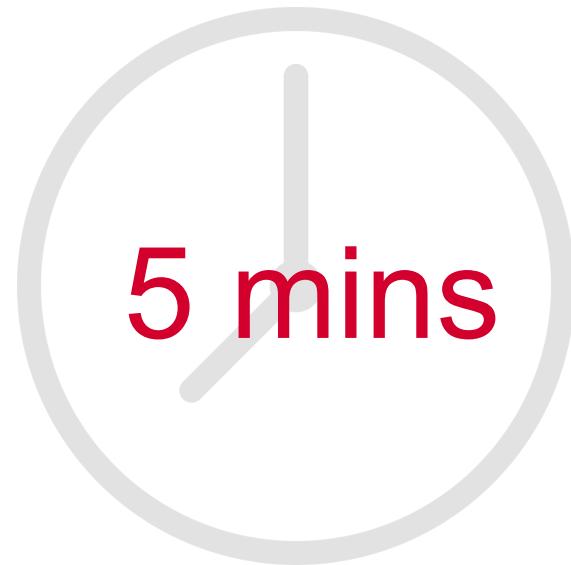
Making Use of a
Provisioner



Q & A



BREAK

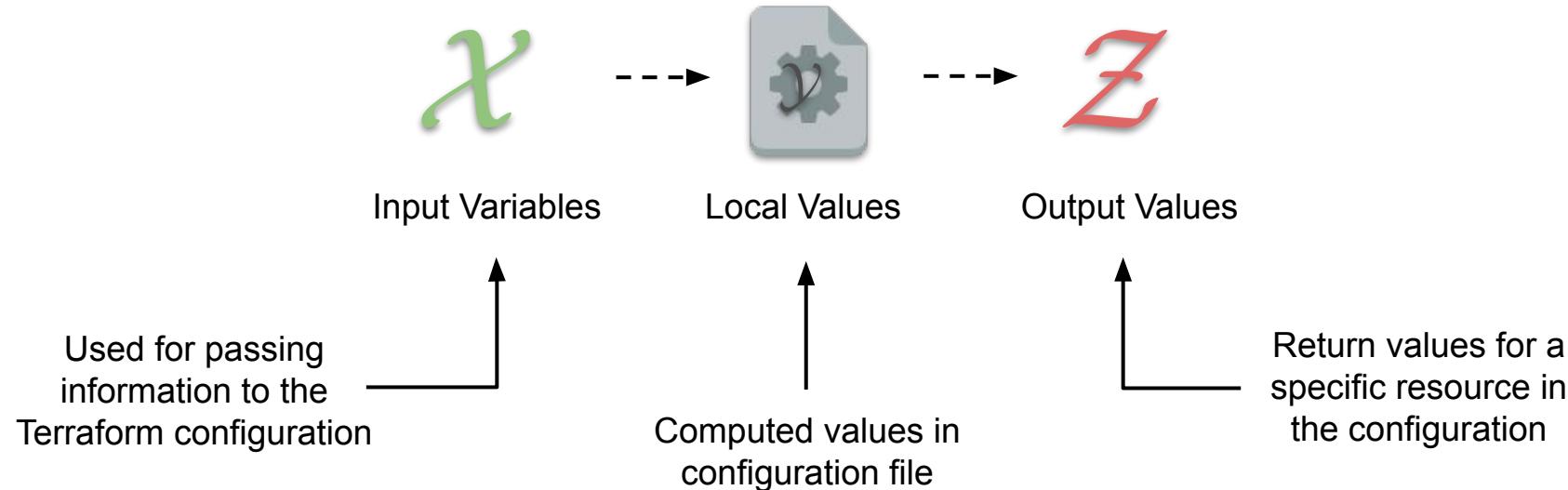


Input Variables, Local/Output Values, and Data Sources

Resource Addressing, Input and Local Variables,
Data Types, Output Values, Resource
Dependencies

Defining Named Values

Concepts for requesting, referencing, and publishing values

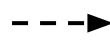


Use Cases for Named Values

Understanding when to use which type



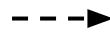
Input Variables



“I want to capture values from the CLI when the end user invokes Terraform and use it in my configuration.”



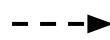
Local Values



“I want to create a variable, assign a value, and reuse it in my configuration similar to a constant.”



Output Values



“I want to render a runtime value in the console output or use it as an input for a resource or module.”



File Naming Conventions

Terraform will automatically resolve those files



Input Variables

variables.tf



Local Values

locals.tf



Output Values

outputs.tf



Resource Addressing

Referencing a variable or resource attribute in a different context

```
# referencing a input variable value  
var.base_image  
  
# referencing a resource attribute value  
aws_instance.example_a.id  
  
# referencing an element in a resource attribute of type list  
azurerm_virtual_network.vnet.subnet[1].id
```



Defining Input Variables

Configuration file can define 0 to many variables

```
variable "ami_id" {
    type = string
    description = "The AMI identifier to use
                    for EC2 instance."
}

variable "availability_zone_names" {
    type     = list(string)
    default = ["us-west-2"]
}
```



Input Variable Arguments

All arguments are optional

Argument	Description
default	Value to be used if not provided
type	Accepted type for values assigned to variable
description	End user description that explains purpose and kind of value
validation	Validation rules applied to provided value
sensitive	Obfuscates sensitive information in CLI output
nullable	Defines if assigned value can be <code>null</code>



Input Variable Data Types

If no type constraint is set then a value of any type is accepted

- **Primitive:** string, number, bool
- **Collection:** list(type), set(type), map(type)
- **Complex:** object(attribute_name = value, ...), tuple(type, ...)



Referencing Input Variable Values

A variable can be used with the `var.name` notation

```
resource "aws_instance" "example" {
    ami           = var.ami_id      ← string value
    instance_type = var.availability_zone_names[0]
}
```



First element in list



Requesting Variable Input Values

End user has to provide value if no default value has been set

```
$ terraform plan  
var.ami_id ← Variable name  
The AMI identifier to use for EC2 instance.  
  
Enter a value: ami-077ee47512dc6f3ca
```

The diagram illustrates the interaction between Terraform command-line output and user input. A horizontal line with arrows points from the user's input 'Enter a value:' back to the 'var.ami_id' variable definition in the command-line output. Three labels with arrows point to specific parts of the interface: 'Variable name' points to the label 'var.ami_id', 'Description' points to the explanatory text 'The AMI identifier to use for EC2 instance.', and 'End user-provided values' points to the user's input 'Enter a value: ami-077ee47512dc6f3ca'.

Variable name

Description

End user-provided values



Variable Input Values from CLI

-var option when running the plan and apply commands

```
$ terraform apply -var="ami_id=ami-0c55b159cbfafef0"  
...  
$ terraform apply  
-var='availability_zone_names=["us-east-1a", "us-west-1c"]'  
...
```



Variable Input Values from Files

-var-file option can point to a file containing input variables

```
$ terraform apply -var-file="runtime-var.tfvars"
```

```
ami_id = "ami-0c55b159cbfafef0"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c"
]
```

runtime-var.tfvars



Auto-Loading of Variable Files

Standard naming convention will apply

- Files named exactly `terraform.tfvars` or `terraform.tfvars.json`
- Any files with names ending in `.auto.tfvars` or `.auto.tfvars.json`



Injecting Secrets as Inputs

Input variables will be stored in the state in plain text

```
$ export TF_VAR_third_party_pwd=s3cr3t  
$ export TF_VAR_api_key=a8an23sdf023xmkdd
```

main.tf

```
# use variables  
var.third_party_pwd  
var.api_key
```



Validating Input Variables

Built-in functions can be used to implement validation logic

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



EXERCISE

Input Variable
Definition and
Consumption



Defining Local Values

Supports reusability of the same value in a configuration

```
locals {
    some_other = var.from_cli
    default_tags = {
        Organization = "O'Reilly"
        Owner = "Benjamin Muschko"
    }
}
```

Parsed value from
input variable



Referencing Local Values

A local value can be used with the `local.name` notation

```
provider "aws" {
  region = "us-east-2"
  default_tags {
    tags = local.default_tags
  }
}
```



Assigns the local value from
the locals definition block



EXERCISE

Local Value
Definition and
Consumption



Defining Output Values

Make information about infrastructure available on CLI

```
output "example_ip_address" {
    value = aws_instance.example.private_ip
    description = "The private IP address of the
                    main server instance."
}
```

References the
private IP address
attribute of the
AWS instance
named example



Output Values in CLI

Outputs can be queried from the state database (if populated)

```
$ terraform output
Warning: No outputs found
The state file either has no outputs defined, or all the
defined outputs are empty.

$ terraform apply
example_ip_address = "..."

$ terraform output
example_ip_address = "..."
```



EXERCISE

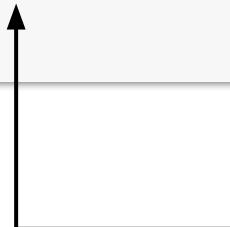
Output Value
Definition and
Rendering



Implicit Dependencies

Using an output value as input for a resource

```
resource "aws_eip" "ip" {
    vpc = true
    instance = aws_instance.example.id
}
```



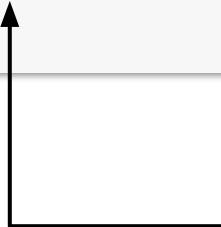
Ensures that EC2 instance
is created before the
elastic IP resource



Explicit Dependencies

Sometimes Terraform cannot determine dependencies

```
module "example_sqs_queue" {  
  source = "terraform-aws-modules/sqs/aws"  
  version = "2.1.0"  
  depends_on = [aws_s3_bucket.example, aws_instance.example_c]  
}
```



Declaration order does not
guarantee execution order



Defining Data Sources

Get information about resources external to Terraform

```
# define data source
data "github_repository_pull_requests" "pull_requests" {
    base_repository = "example-repository"
    base_ref = "main"
    state = "open"
}

# Reference data
data.github_repository_pull_requests.pull_requests.results
```



EXERCISE

Defining and
Consuming a Data
Source



Q & A



BREAK



Iteration Control, Functions, and Expressions

Loops, Dynamic Blocks, Built-In Expressions

Looping Constructs

Terraform provides looping syntax for different use cases

- count argument: Assigns an integer that determines # of loops
- for_each argument: Iterates over each element of a map or set
- for expression: Used to transform elements in a map or set
- dynamic blocks: Similar to a traditional for-loop but uses the dynamic nested block syntax



Count Argument

Count is a simple integer value assignment

```
resource "aws_instance" "app_server" {
  count = 3
  ami   = "ami-0c55b159cbfafef0"
  instance_type = "t2.micro"
  tags  {
    Name = "app_server_${count.index}"
  }
}
```



Current index of loop
can used as variable



For Each Argument

Iteration element can be referenced by key and value attribute

```
resource "azurerm_resource_group" "rg" {
  for_each = {
    a_group = "eastus"
    another_group = "westus2"
  }
  name      = each.key
  location  = each.value
}
```

map data type with 2 entries

For a set data type the key
is the same as value



For Expression

Applies an expression to each element to transform it

```
output "all_tags" {
    value = [for key, value in var.tags :
              upper(key) => upper(value)]
}
```

← Iterate over
elements in a
map

Transforms key and
value with function
and outputs a map



Dynamic Block

Helps with implementing DRY for more elaborate configurations

```
resource "aws_security_group" "security" {
  ...
  dynamic "ingress" {
    for_each = local.ingress_rules
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Local variable pointing to a set of tuples with port numbers and descriptions

Configure the whole “block” for each iteration the set



EXERCISE

Declaring a Loop
Sourced from an
Input Variable



Built-In Functions

Functionality for transforming, combining, and formatting data

- Grouped into higher-level categories e.g. collections, type conversion, date and time ([docs](#))
- Syntax in code: `function(argument, ...)`
- Functions can be combined with each other by nesting them
- Terraform provides a console to try out functions against state data



Function Examples

Available functions are vast, suitable for most common use cases

```
# return largest value in a set  
max([22, 5, 99])  
  
# base64-encode a string value  
base64encode("Hello World")  
  
# extract a substring from a given string  
substr("hello world", 1, 4)
```



Terraform Console

Test functions against the current state data (and locks it)

```
$ terraform console
> var.instance_count
3
> range(0, var.instance_count)
tolist([
  0,
  1,
  2,
])
```

Get the value of the input variable `instance_count`

Execute the `range` function against the variable



EXERCISE

Using Built-In
Functions to
Transform Data



Interacting with Modules

Module Structure and Syntax, Inputs & Outputs

What is a Module?

Reusable functionality in the world of Terraform

- Usually defines inputs, resources/data sources, and outputs (though they are all optional)
- Modules are versioned (standardized on semantic versioning)
- The `init` command will download remote modules and store them in the `.terraform` directory



Module Sources

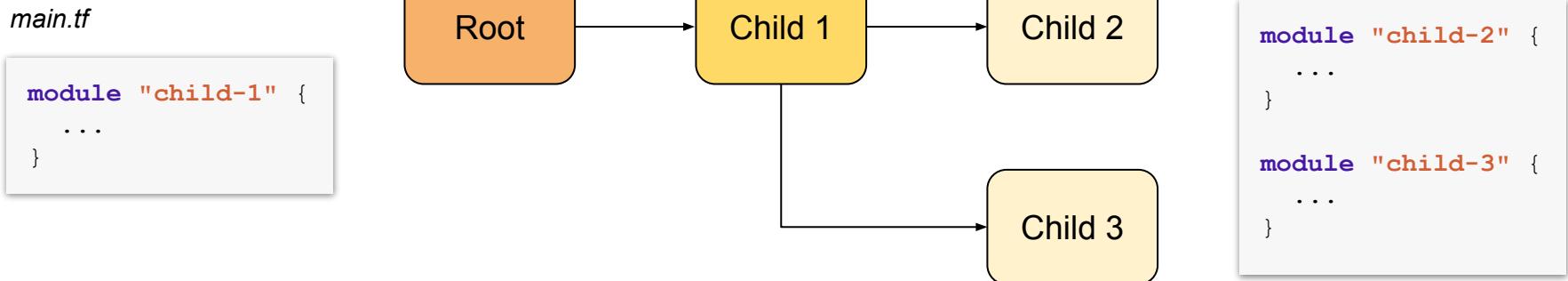
A module can be sourced from different locations

- A local path to a directory present on disk
- A registry like the [Terraform registry](#)
- A Git repository e.g. GitHub or BitBucket
- A HTTP(S) URL that follows a provider protocol



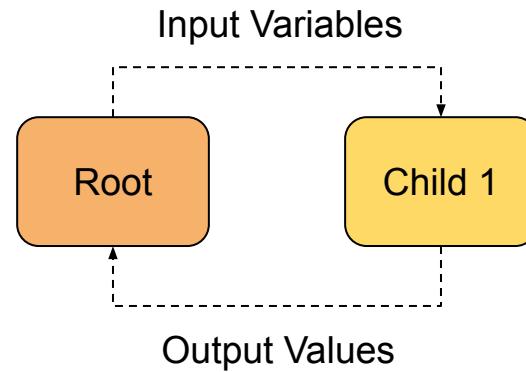
Root and Child Modules

Modules can be consumed on any level of the hierarchy



Module Data Exchange

Modules only communicate via inputs and outputs



Module Example

Organized in a subdirectory, and defined by .tf files

modules/eks/main.tf

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

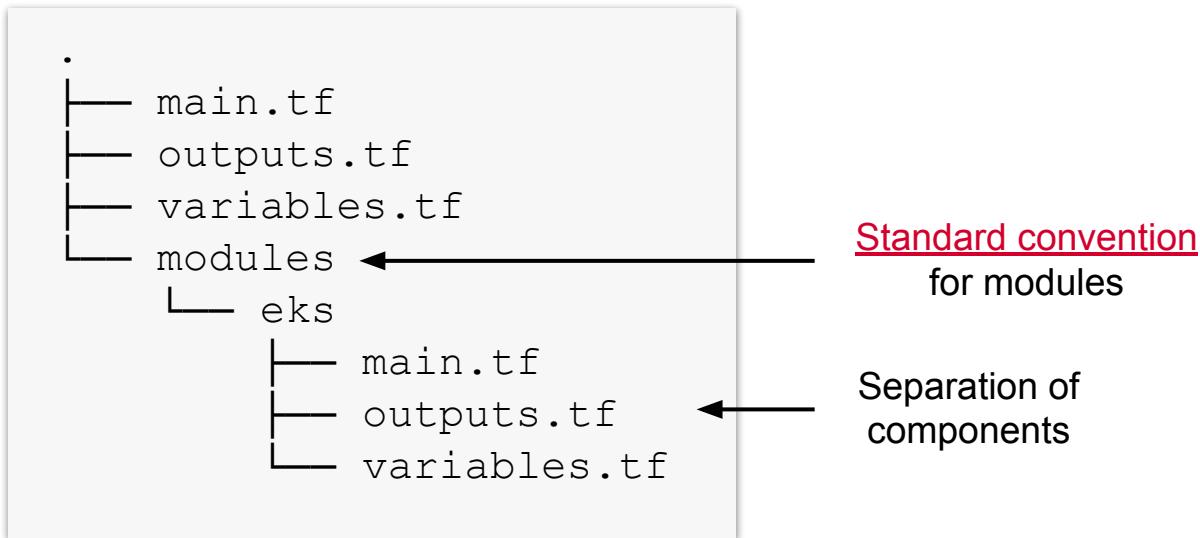
resource "aws_eks_cluster" "eks_example" {
    name = "${var.cluster_name}-eks"
    ...
}

output "endpoint" {
    value = aws_eks_cluster.eks_example.endpoint
}
```



Module Structure

Organized in a subdirectory, and defined by .tf files



Consuming a Local Module

References by source, no version needs to be defined

main.tf

```
module "aws_eks_cluster" {  
  source = "./modules/eks"  
  cluster_name = "my_cluster"  
}  
  
resource "..." {  
  endpoint =  
    module.aws_eks_cluster.endpoint  
}
```

Location of module in directory structure

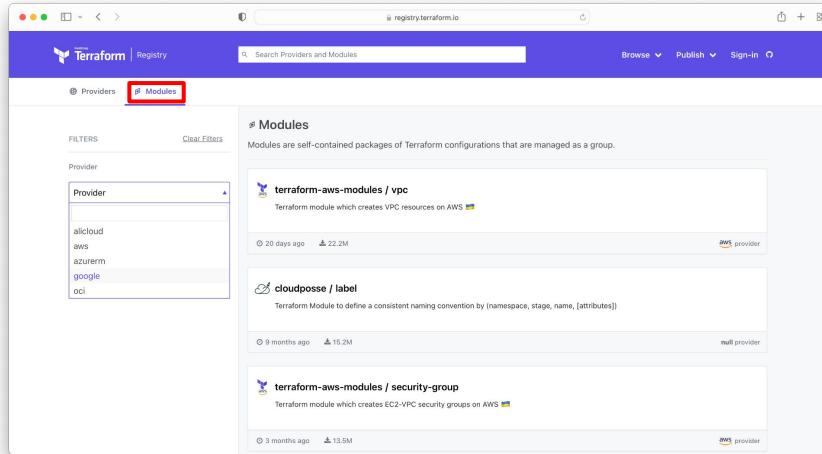
Providing the value for the input variable

Accessing the output value from local module



Exploring the Module Registry

Similar to providers, modules are available Terraform registry



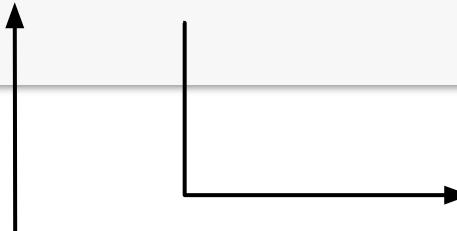
<https://registry.terraform.io/browse/modules>



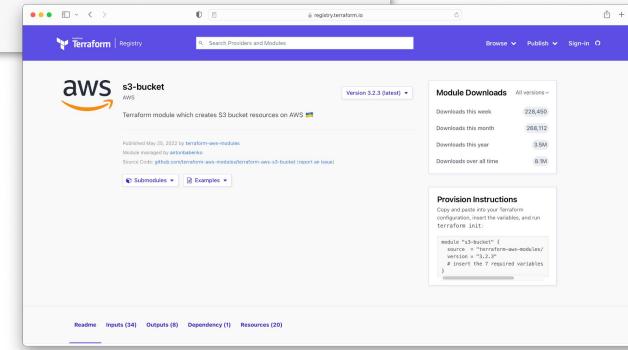
Consuming a Public Module

Define source and version

```
module "s3-bucket" {  
  source = "terraform-aws-modules/s3-bucket/aws"  
  version = "3.2.3"  
}
```



Follows the same
conventions and notation as
provider versions



EXERCISE

Implementing and
Using a Custom
Module



Q & A



BREAK

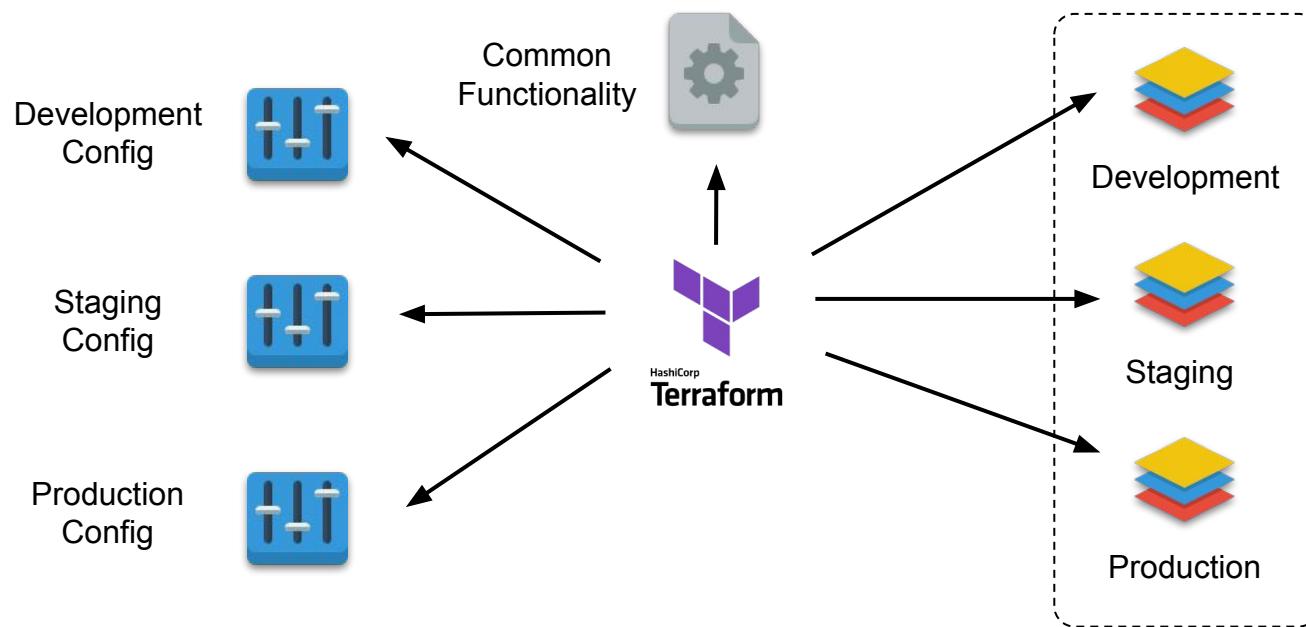


Managing Multiple Runtime Environments

Workspaces, Commands, Variable Values,
Separated State

Multiple Environments

Commonalities and differences



Handling Environments

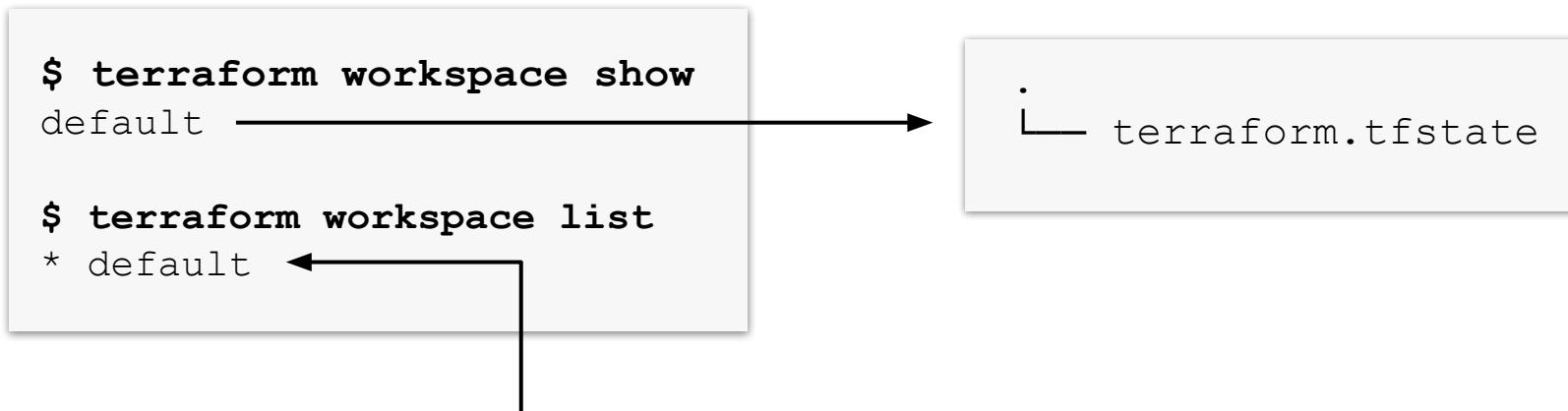
A workspace manages requirements

- Shared configuration, different input variable values per context
- Unique credentials per environment to decrease security risks
- Holding unique state per environment
- Increased maintenance effort and potential complexity to end users



The Default Workspace

Cannot be deleted and uses state file in root directory



Marks the selected workspace with an asterisk character



Creating a New Workspace

Built-in command that creates a dedicated state folder

```
$ terraform workspace new development
```

```
Created and switched to workspace "development"!
```

Every workspace
has its own folder



```
.
└── terraform.tfstate.d
    └── development
```



Workspace Variable Values

Per-workspace configuration needs to be resolvable



Development



Staging



Production

Variable	Value
instance_type	t2.nano
instance_count	1

Variable	Value
instance_type	t2.micro
instance_count	2

Variable	Value
instance_type	t2.small
instance_count	4



Defining a Workspace Value

Specify a variable of type map in locals.tf file

```
locals {  
    ec2_instance_type = {  
        development = "t2.nano"  
        staging = "t2.micro"  
        production = "t2.small"  
    }  
  
    ec2_instance_count = {  
        ...  
    }  
}
```

← Assign key-value pairs per environment

← Repeat for every workspace-specific variable



Resolving a Workspace Value

Use the variable `terraform.workspace` to select value

```
resource "aws_instance" "web_server" {  
    ami = "ami-0c55b159cbfafef0"  
    instance_type = local.ec2_instance_type[terraform.workspace]  
    instance_count = local.ec2_instance_count[terraform.workspace]  
}
```

Selects value in
corresponding map
variable



Switching Between Workspaces

Use `select` command to target a different environment

```
$ terraform workspace list
  default
  development
* production
  staging

$ terraform workspace select staging
Switched to workspace "staging".

$ terraform workspace show
staging
```



Deleting a Workspace

Select different workspace first, then run delete command

```
$ terraform workspace select development  
Switched to workspace "development".
```

```
$ terraform workspace delete staging  
Deleted workspace "staging"!
```

Removes workspace
folder + state file

The diagram illustrates the deletion of a workspace folder. On the left, a text box contains the command '\$ terraform workspace delete staging' and its output 'Deleted workspace "staging"!'. An arrow points from this text box to a folder structure on the right. The folder structure shows a root folder with two subfolders: 'development' and 'production'. The 'development' folder contains a file named 'terraform.tfstate.d'. A small red circular icon is located in the bottom right corner of the slide.

```
.  
└── terraform.tfstate.d  
    ├── development  
    └── production
```



EXERCISE

Managing Multiple
Environments With
Workspaces



Implementing and Maintaining State

Local vs. Remote Backend, State Locking, Secret Management

Holding State

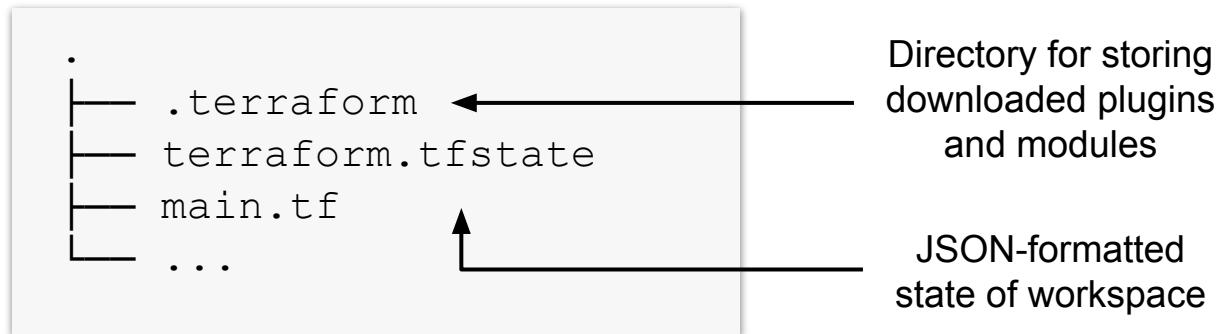
Allows for comparing desired with current infrastructure definition

- Terraform only creates/updates what has actually changed
- Enabler for improving performance
- The commands `plan` and `apply` interact with the state
- Local or remote backend for different use cases



The Default Local Backend

Stored in the `terraform.tfstate` file in workspace directory



Configuring the Local Backend

Default file & location can be changed by CLI or configuration

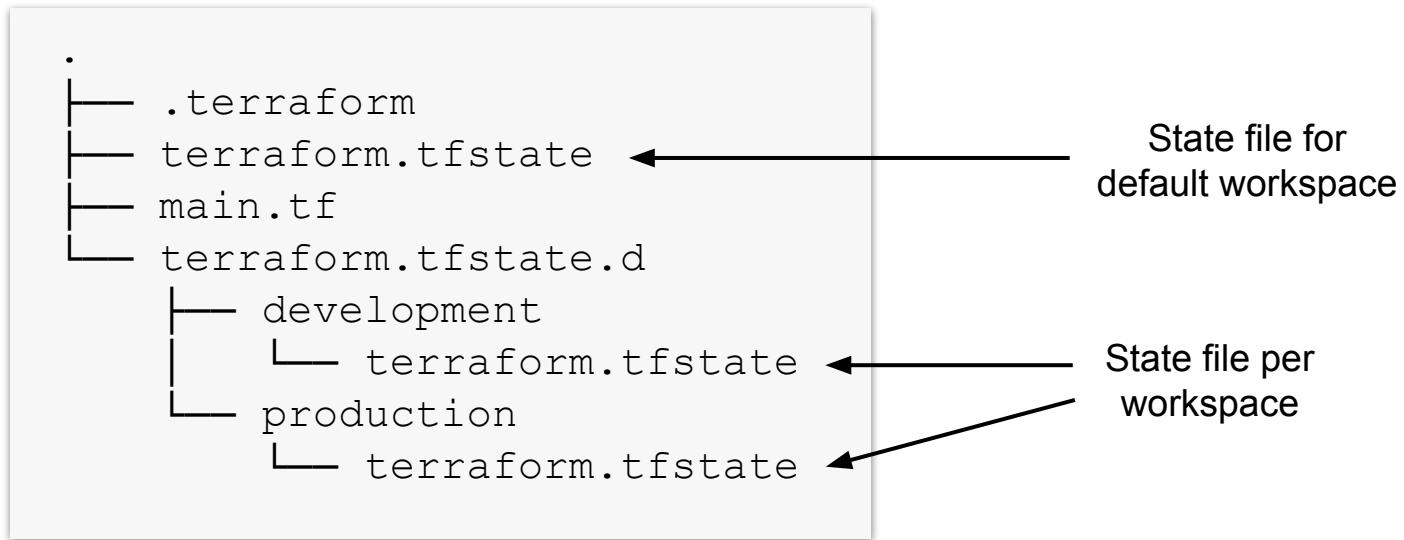
```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

For ad-hoc tasks, you can use the `-state` command line option



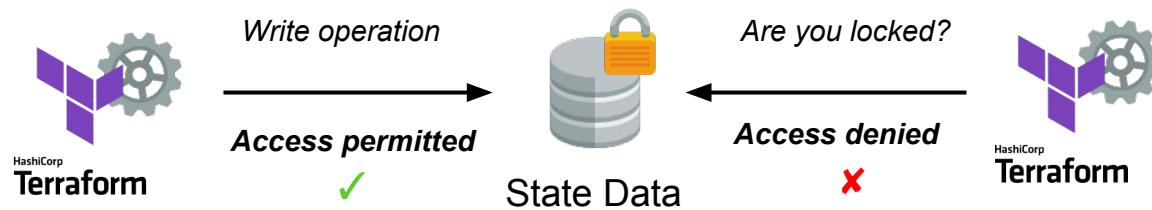
Local State File Per Workspace

State per environment is separated by subdirectory



State Locking

Ensures that state does not corrupted by concurrent access



Lock management can be circumvented using the `-lock=false`.

Caution!



Remote Backends

Optimized for access by multiple users

- Never stores state data on local disk, only loaded into memory and flushed after the operation
- Locking is established on remote backend implementation
- Terraform currently supports different backend [implementations](#)
- Enhanced backends (e.g. Terraform Cloud and Enterprise) run Terraform operation on remote service



Remote Backend Authentication

Credentials shouldn't be configured in .tf file

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Credentials should be provided as environment variables to avoid sending them as plain text over the wire or committing them to version control



Partial Backend Configuration

Information can be provided from the CLI with `init` command

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "company"  
  
    workspaces {  
      name = "my-app-prod"  
    }  
  }  
}
```

```
$ terraform init  
-backend-config=organization=company
```



Secret Management & State

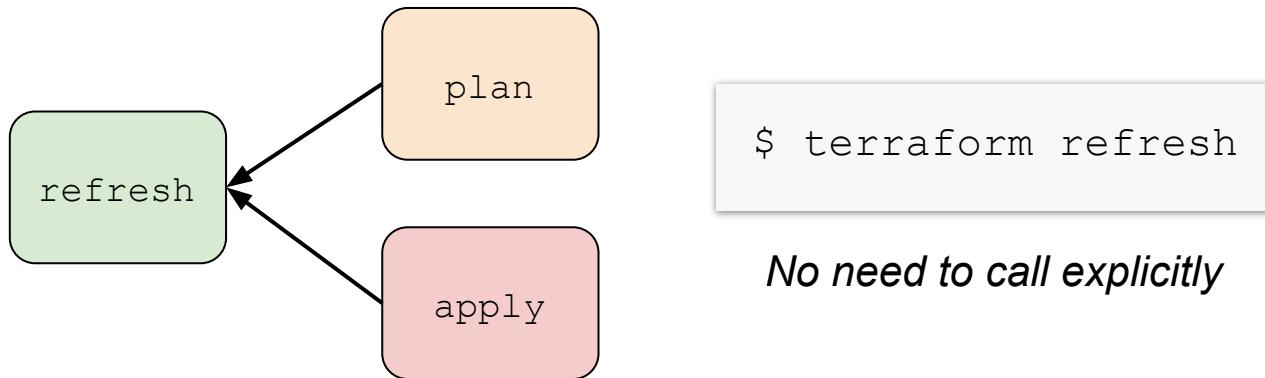
State data is not encrypted and shouldn't store sensitive data

- Sensitive data can be provided with environment variables for some remote backend implementations
- Enhanced backends send and store state information in encrypted form
- Credential-management platforms like [HashiCorp Vault](#) can integrated as a viable alternative for storing and providing credentials



Managing Resource Drift

Ensures that state synchronizes with deployed infrastructure



The `plan` and `apply` command automatically refresh the state.



Example Refresh Output

Calculated differences between state and cloud are rendered

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.example: Refreshing state... (ID: i-011a9893eff09ede1)
```

```
-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
~ update in-place
```

```
Terraform will perform the following actions:
  ~ aws_instance.example
    tags.drift_example: "v2" => "v1"
```

Output will list actions
when `apply` command
is performed



Resource Drift Output Symbols

Refresh output shoulds differences and their actions up apply

Symbol	Meaning
+	Create
-	Destroy
-/+	Replace (destroy and then create, or vice-versa if create-before-destroy is used)
~	Update in-place
<=	Read (only applies to data sources)



Deprecation of refresh Command

The command will be removed in a future Terraform version

- The refresh command means the same apply -refresh-only -auto-approve. Therefore, the state will be overwritten. You do have a backup file with the file extension .backup.
- Invalid provider credentials or wrong credentials for a region can lead to an incorrect state.
- It's recommended to run plan/apply with -refresh-only option. The apply explicitly asks for confirmation before overwriting the state file. More information on this [blog post](#).



EXERCISE

Managing Resource
Drift



Advanced Terraform Workflows

fmt, validate, import, state, Enabling verbose
logging

Validating Configuration Files

Check for syntactical errors in configuration files of a directory

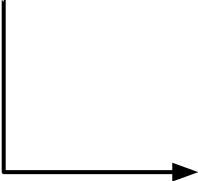
- Functionality provided by the validate command
- Does not consider provided variables or state
- Does not make a guarantee on successful deployment
- Should be run before plan/apply commands and is a good fit for CI/CD pipelines



Validation Example

Renders errors on terminal output

```
instance_type = ["t2.micro"]
```



```
$ terraform validate
Error: Incorrect attribute value type
on main.tf line 16, in resource "aws_instance" "app_server":
16:   instance_type = ["t2.nano"]

Inappropriate value for attribute "instance_type": string required.
```



Formatting Configuration Files

Align configuration with standard style conventions

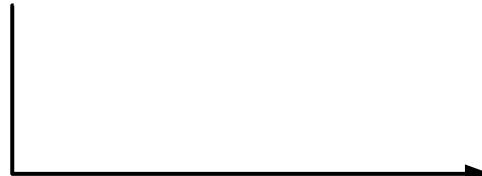
- Functionality provided by the `fmt` command
- Applies to `.tf` and `.tfvars` files in the current directory, use `-recursive` to apply formatting to all subdirectories
- Other CLI options are available: `-list=false`, `-write=false`, `-diff`, `-check`
- Should be performed before committing to version control



Reformatting Example

Aligns multiple attribute assignments for a resource

```
ami = "ami-0c55b159cbfafef0"  
instance_type = "t2.micro"
```



```
ami          = "ami-0c55b159cbfafef0"  
instance_type = "t2.micro"
```



EXERCISE

Validating and
Formatting
Configuration Files



Importing Existing Infrastructure

Bring manually created infrastructure under Terraform's control

- Functionality provided by the `import` command
- Loads supported resources into your Terraform workspace's state
- Does not automatically generate the configuration to manage the infrastructure
- Importing infrastructure is a [multi-step process](#)



Identifying the Resource

Every resource to be imported needs to have an identifier

Instances (1) Info											
		Actions Launch instances									
<input type="button" value="Search"/> <input type="text" value=""/>		Connect Instance state Actions Launch instances									
<input type="button" value="Instance state = running"/> X Clear filters											
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 IP	Elastic IP		
-	i-094fd5c99f77c7afe	Running	t2.nano	2/2 checks passed	No alarms	+ us-west-2c	ec2-54-214-82-80.us-w...	54.214.82.80	-		



For an EC2 instance,
the resource identifier
is the *instance ID*



Adding a Resource Config Block

The `import` command will attach state to configuration

```
resource "aws_instance" "web_server" {}
```

Do not add any specific
configuration yet



Running the Import Command

Provide the configuration ID + resource ID

```
$ terraform import aws_instance.web_server i-094fd5c99f77c7afe
```



Configuration ID



Resource ID



Check Imported State

Render the resource state that could be imported

```
$ terraform show
```



Manually Add Configuration

Fill in missing information that couldn't be derived

```
resource "aws_instance" "web_server" {  
    ...  
}
```

Add the
configuration of
the resource



Plan and Apply Changes

Iteratively plan, add configuration, and finally apply

```
$ terraform plan  
$ terraform apply
```



Interacting with Terraform State

Fine-grained interactions with state backend

- Manual updates to the state document, local or remote, should be avoided
- Functionality provided by the `state` command
- Subcommands available for listing resources, removing resources, moving items, pulling/pushing the state between local and remote backend, and rendering the current state



Listing Current Resources

“What resources are currently managed by Terraform?”

```
$ terraform state list
aws_instance.web_server
aws_ebs_volume.web_server_vol
```

See [command details](#)



Rendering Resource Attributes

“Show me the current configuration of this resource.”

```
$ terraform state show aws_instance.web_server
resource "aws_instance" "web_server" {
    ...
}
```

See [command details](#)



Renaming a Resource

“Renaming a resource without deleting/recreating it first”

```
$ terraform state mv aws_instance.web_server  
aws_instance.app_server  
Move "aws_instance.web_server" to "aws_instance.app_server"  
Successfully moved 1 object(s).
```

See [command details](#)



Remove an Item from the State

“I want to manage a resource outside of Terraform”

```
$ terraform state rm aws_instance.web_server
Removed aws_instance.web_server
Successfully removed 1 resource instance(s).
```

```
# Remove the resource from the configuration
$ terraform plan
$ terraform apply
```

See [command details](#)



EXERCISE

Interacting with the
State From the CLI



Enabling Verbose Logging

Rendering detailed information in error situations

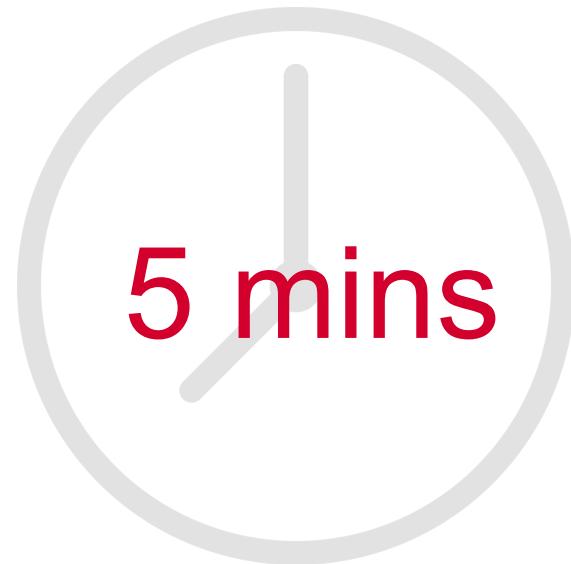
- Terraform writes logs to standard error and offers the following log levels: TRACE, DEBUG, INFO, WARN, and ERROR
- The log level can be set via the environment variable `TF_LOG`
- The log level for Terraform itself and provider plugins can be controlled with the environment variable `TF_LOG_CORE` and `TF_LOG_PROVIDER`



Q & A



BREAK



Understand Terraform Cloud Capabilities

How Does it Help in an Enterprise Setting?

What is Terraform Cloud?

Commercial, cloud-hosted distribution of Terraform

- Enhanced features for team collaboration and enterprise-level infrastructure management (e.g. security and compliance)
- Hosted service available at <https://app.terraform.io>
- Terraform Enterprise is a self-hosted distribution of Terraform Cloud



Feature: Account Management

Authentication and authorization

- A company can model organizational structure with the concepts *users, teams, and organizations*
- Permissions can be defined to control authorization of operations
- **Benefit:** Added security by only allowing authenticated access, User can only execute operations with granted permissions, Single Sign On (SSO) integration



Feature: VCS Integration

Terraform workflow integration as part of CI/CD automation

- Supports GitHub, GitLab, BitBucket, and Azure DevOps
- Automatic execution of a Terraform workflow upon commit
- Simplified release process of Terraform modules
- **Benefit:** Simplified setup of automatic execution for typical Terraform workflows against code changes without a lot of manual intervention



Feature: Workspaces

Not to be confused with Terraform OSS workspace

- Terraform Cloud and OSS uses the same terminology, however, the meaning for a *workspace* is different
- In Terraform Cloud a workspace is associated with a VCS repository
- Workspace settings include variable values, state, and secrets, and historical logs
- **Benefit:** Avoid local state file(s) and centralize state on remote



Feature: Config. Compliance

Enforced compliance as part of the plan-apply workflow

- Sentinel is a compliance framework for centralizing company rules as code and enforcing them
- Triggers after terraform plan and before terraform apply commands and fails command if configuration doesn't follow policies
- **Benefit:** Standardizing on resource naming conventions or validating attribute values



Feature: Private Registry

Added security by only allowing members to access registry

- Enterprises can host their providers and modules in a private registry (only accessible to members of the organization)
- Includes functionality like versioning, and search capabilities like you already know from the [public Terraform registry](#)
- **Benefit:** Binaries cannot be modified by malicious attacker, attacker cannot publish a new version of binaries that contains malicious code



Other Notable Features

Refer to the [documentation](#) for more information

- API access: Ability to run Terraform operations via a RESTful API
- Remote operations: Consistent and reliable run environment with the help of virtual machines
- Cost estimation: Calculates expected cost when run on cloud provider like AWS, GCP, and Azure



Q & A



EXERCISE

Apache Web Server
on AWS Accessible
via the Internet



Summary & Wrap Up

Last words of advice...



Thank you

