O'REILLY®

HashiCorp Certified: Terraform Associate Certification Crash Course

Curriculum Version 004

About the trainer

bmuschko@automatedascent.com

benjaminmuschko

bmuschko

AUTOMATED ASCENT

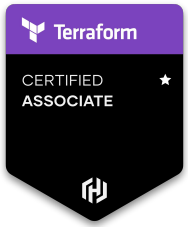
automatedascent.com

Certification Exam

Objectives, Curriculum, Prerequisites, Learning Resources

Exam Objectives

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

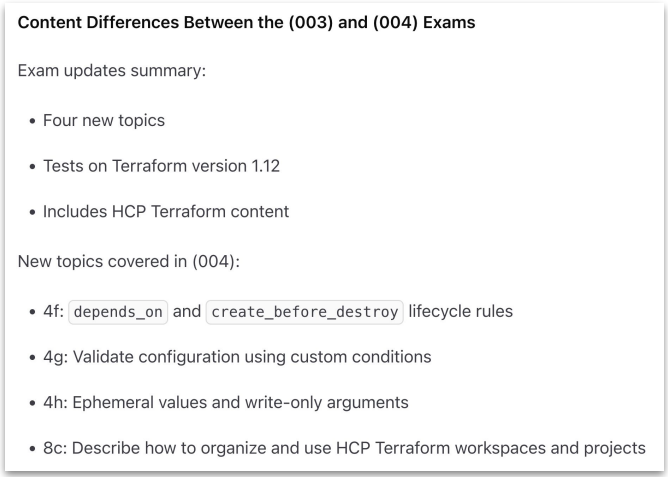


The certification program allows users to demonstrate their competence in a multiple choice test.

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

Curriculum Changes

Version 004 replaced 003 in January 2026



Content Differences Between the (003) and (004) Exams

Exam updates summary:

- Four new topics
- Tests on Terraform version 1.12
- Includes HCP Terraform content

New topics covered in (004):

- 4f: `depends_on` and `create_before_destroy` lifecycle rules
- 4g: Validate configuration using custom conditions
- 4h: Ephemeral values and write-only arguments
- 8c: Describe how to organize and use HCP Terraform workspaces and projects

https://www.hashicorp.com/certification/terraform-associate-004

The Curriculum

1	**Infrastructure as Code (IaC) with Terraform**
1a	Explain what IaC is
1b	Describe the advantages of IaC patterns
1c	Explain how Terraform manages multi-cloud, hybrid cloud, and service-agnostic workflows
2	**Terraform fundamentals**
2a	Install and version Terraform providers
2b	Describe how Terraform uses providers
2c	Write Terraform configuration using multiple providers
2d	Explain how Terraform uses and manages state
3	**Core Terraform workflow**
3a	Describe the Terraform workflow
3b	Initialize a Terraform working directory
3c	Validate a Terraform configuration
3d	Generate and review an execution plan for Terraform
3e	Apply changes to infrastructure with Terraform
3f	Destroy Terraform-managed infrastructure
3g	Apply formatting and style adjustments to a configuration

The Curriculum

3	**Core Terraform workflow**
3a	Describe the Terraform workflow
3b	Initialize a Terraform working directory
3c	Validate a Terraform configuration
3d	Generate and review an execution plan for Terraform
3e	Apply changes to infrastructure with Terraform
3f	Destroy Terraform-managed infrastructure
3g	Apply formatting and style adjustments to a configuration
4	**Terraform configuration**
4a	Use and differentiate `resource` and `data` blocks
4b	Refer to resource attributes and create cross-resource references
4c	Use variables and outputs
4d	Understand and use complex types
4e	Write dynamic configuration using expressions and functions
4f	Define resource dependencies in configuration
4g	Validate configuration using custom conditions
4h	Understand best practices for managing sensitive data, including secrets management with Vault

The Curriculum

5	**Terraform modules**
5a	Explain how Terraform sources modules
5b	Describe variable scope within modules
5c	Use modules in configuration
5d	Manage module versions
6	**Terraform state management**
6a	Describe the local backend
6b	Describe state locking
6c	Configure remote state using the backend block
6d	Manage resource drift and Terraform state
7	**Maintain infrastructure with Terraform**
7a	Import existing infrastructure into your Terraform workspace
7b	Use the CLI to inspect state
7c	Describe when and how to use verbose logging
8	**HCP Terraform**
8a	Use HCP Terraform to create infrastructure
8b	Describe HCP Terraform collaboration and governance features
8c	Describe how to organize and use HCP Terraform workspaces and projects
8d	Configure and use HCP Terraform integration

Candidate Prerequisites

Basic Terminal Skills

Basic understanding of on premises and cloud architecture

Preparing for the Exam

Documentation is not allowed during the test

- Learning Path: https://developer.hashicorp.com/terraform/tutorials/certification-004/associate-study-004
- Exam Content List: https://developer.hashicorp.com/terraform/tutorials/certification-004/associate-review-004

Preparing for the Exam

Documentation is not allowed during the test

- Sample Questions: https://developer.hashicorp.com/terraform/tutorials/certification-004/associate-questions-004

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

- Business source licensed (BSL) 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)

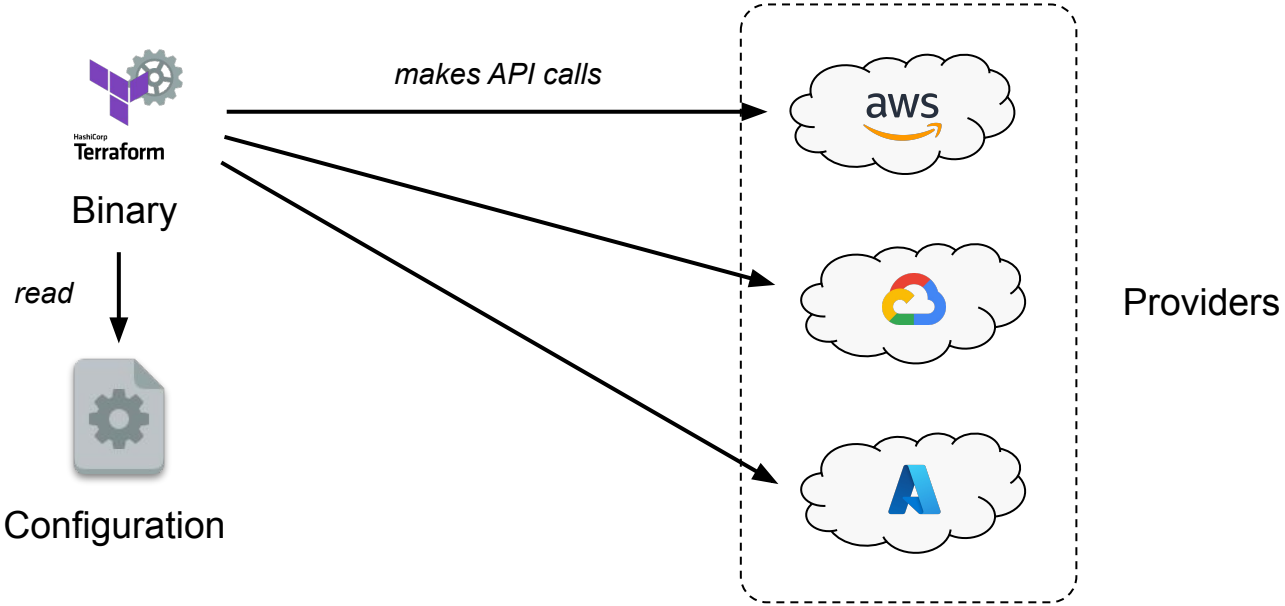
How Does Terraform Work?

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-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

```
resource "aws_instance" "example" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

State Storage

Where and how is state stored and updated?

- Stores state in “internal database”, each resource is represented by an 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 OpenTofu

Differences and use cases

- In August 2023, HashiCorp changed Terraform's license from MPL to BSL. OpenTofu emerged as a community fork. The Terraform Associate certification is **not** about OpenTofu.

- Key differences emerging
  - OpenTofu added native state encryption
  - Feature development is diverging over time
  - Some newer Terraform features may not appear in OpenTofu (and vice versa)

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 as 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 HCP Terraform

Q & A

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 v1.14.1
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

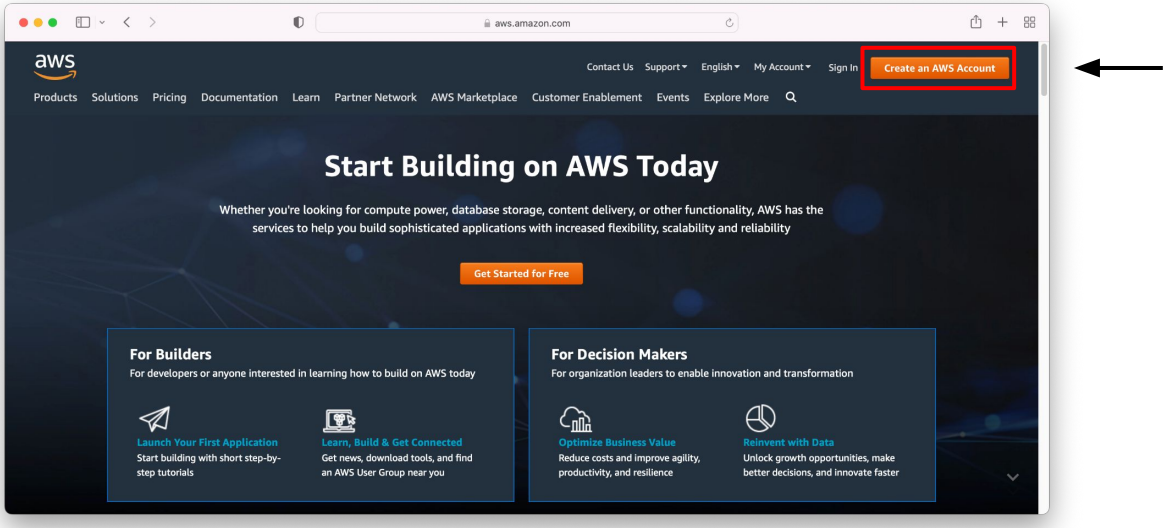
tfswitch lets you manage and use versions in parallel


```
$ 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
```

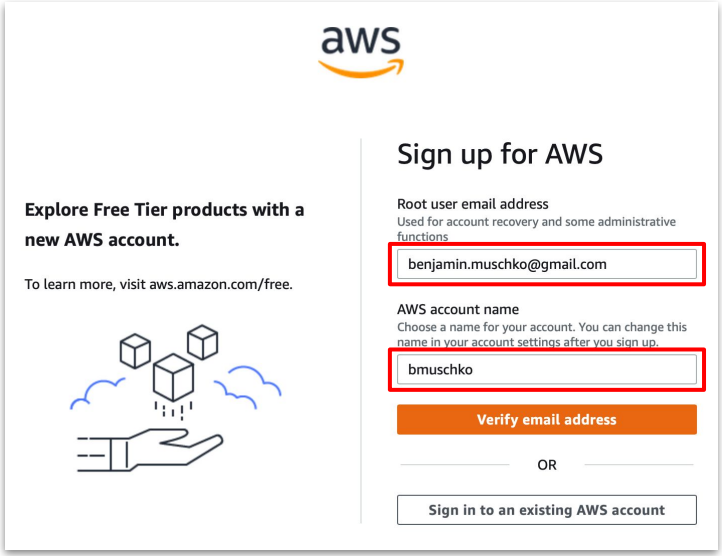
Setting Up an AWS Account

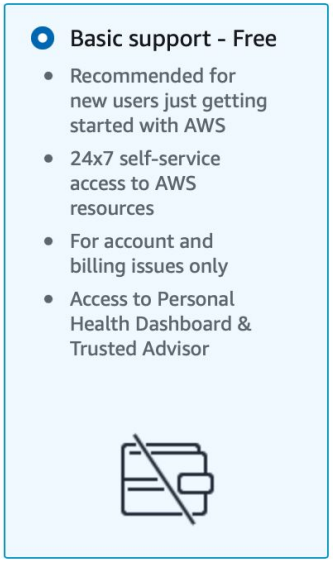
Create new account at https://aws.amazon.com

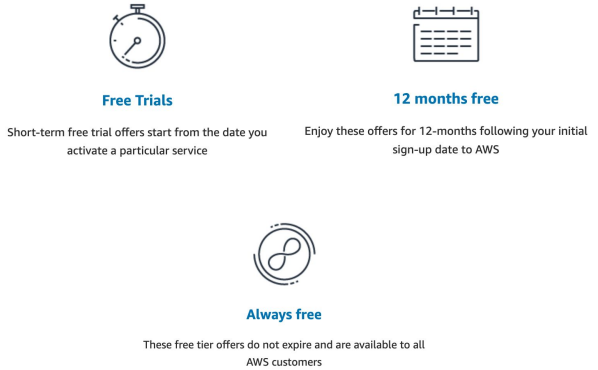


Setting Up an AWS Account

Free tier is sufficient for exercises in this course

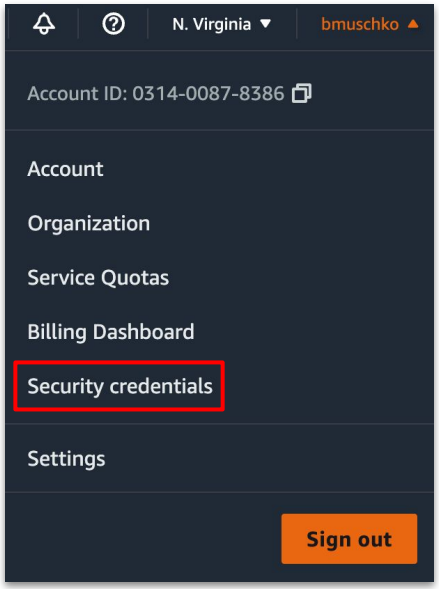


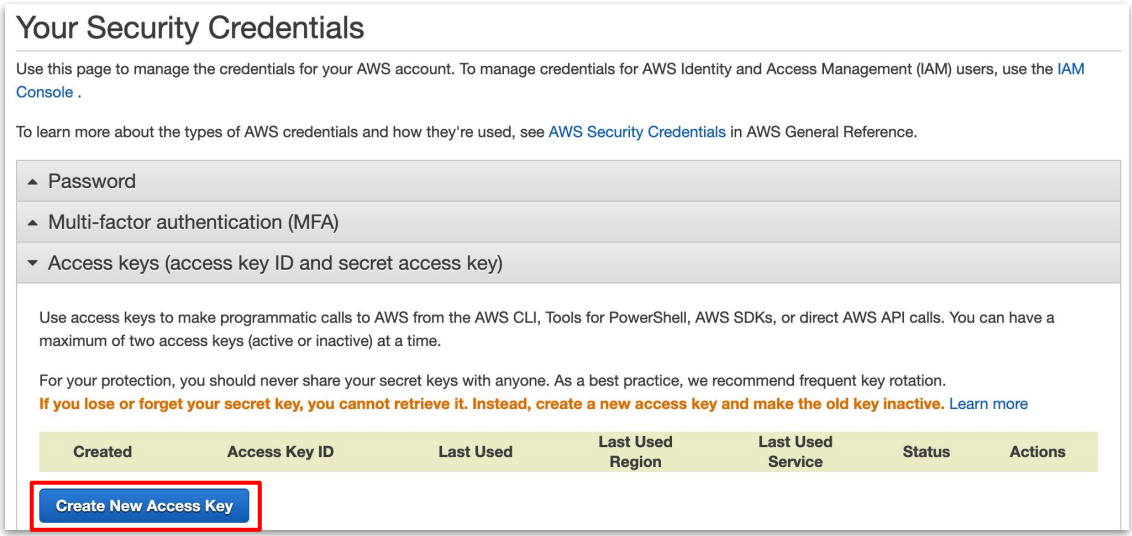




Retrieving AWS Credentials

Create a new access key and store in safe place





Setting Provider Credentials

In current shell or from typical credentials location

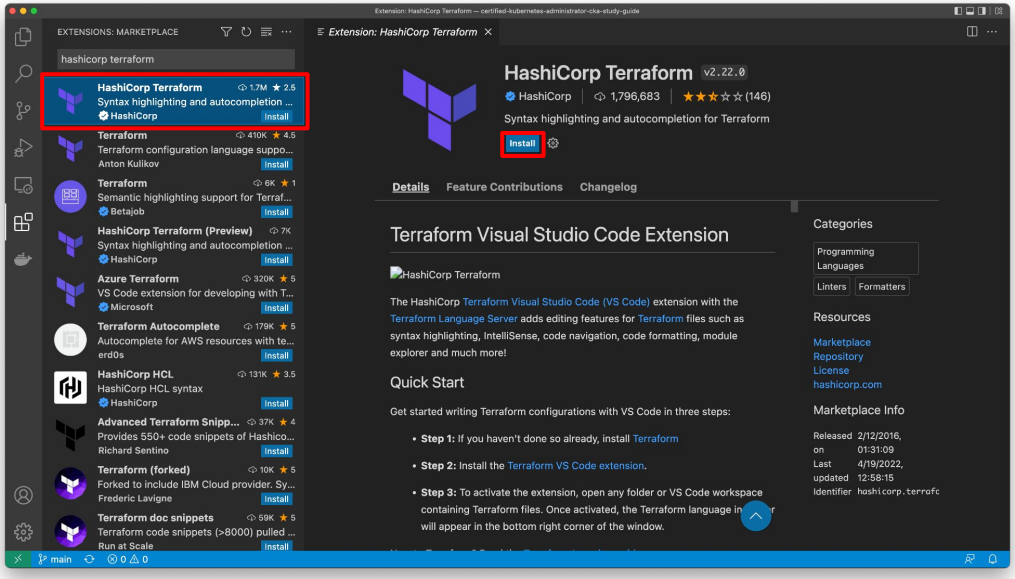
```
$ export AWS_ACCESS_KEY_ID=<access-key-id>
$ export AWS_SECRET_ACCESS_KEY=<secret-access-key>
```

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

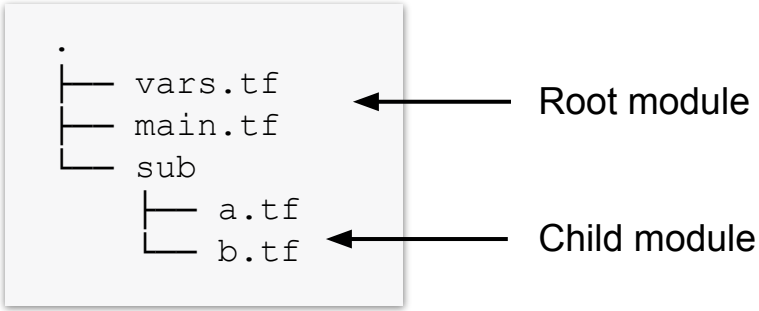
Getting Started With Terraform

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

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL NAME>" {
  <IDENTIFIER> = <EXPRESSION>
}
```

Block: A container for defining the configuration of an object delimited by curly braces (can have nested blocks)

Argument: A key-value pair defined within a block

Expression: Computed values assigned to an argument

Syntax Constraints

Just of a couple of little gotchas to look out for

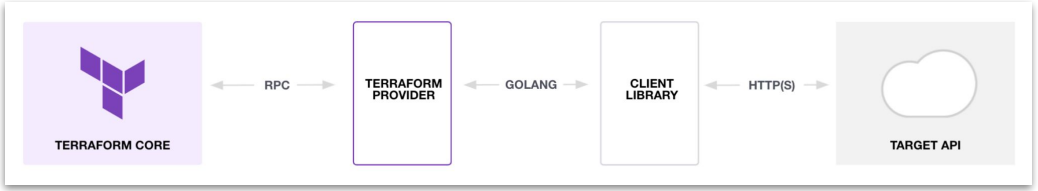
- *Identifiers* can contain letters, digits, underscores (_), and hyphens (-). The first character of an identifier must not be a digit, to avoid ambiguity with literal numbers.
- Comments can be defined by `#`, `//`, or `/* */`.
- Configuration files must always be UTF-8 encoded

Plugin-Based Architecture

Core binary is small, providers are developed as plugins

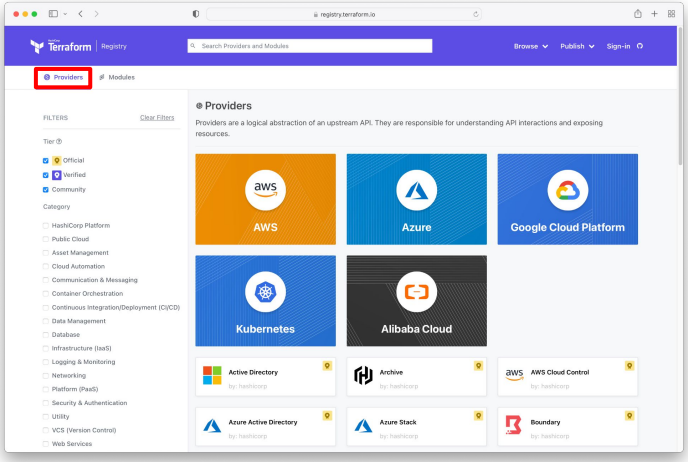
- *Terraform Core:* The binary that communicates with plugins to manage infrastructure resources

- *Terraform plugins:* Executable binaries written in Go that communicate with Terraform core over an RPC interface



Exploring the Provider Registry

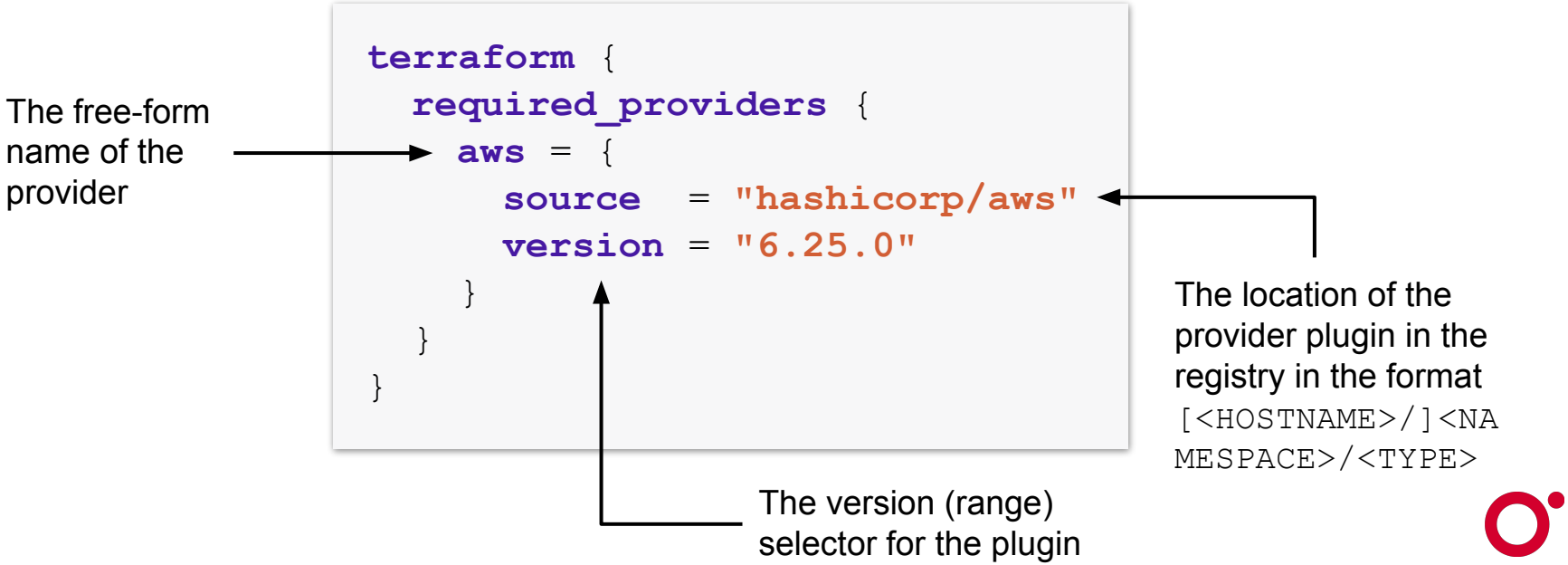
Central location for provider plugins and their documentation



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

Defining Required Provider(s)

Sets up the properties for all providers of the assigned name



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
`>= 1.2.1`	Great than or equal to the version `1.2.1`
`<= 1.2.1`	Less than or equal to the version `1.2.1`
`~> 1.2.1`	Any version in the `1.2.x` range up to the next minor version (`1.3.0`)
`>= 1.2.1, <= 1.5.0`	Any version between `1.2.1` and `1.5.0`

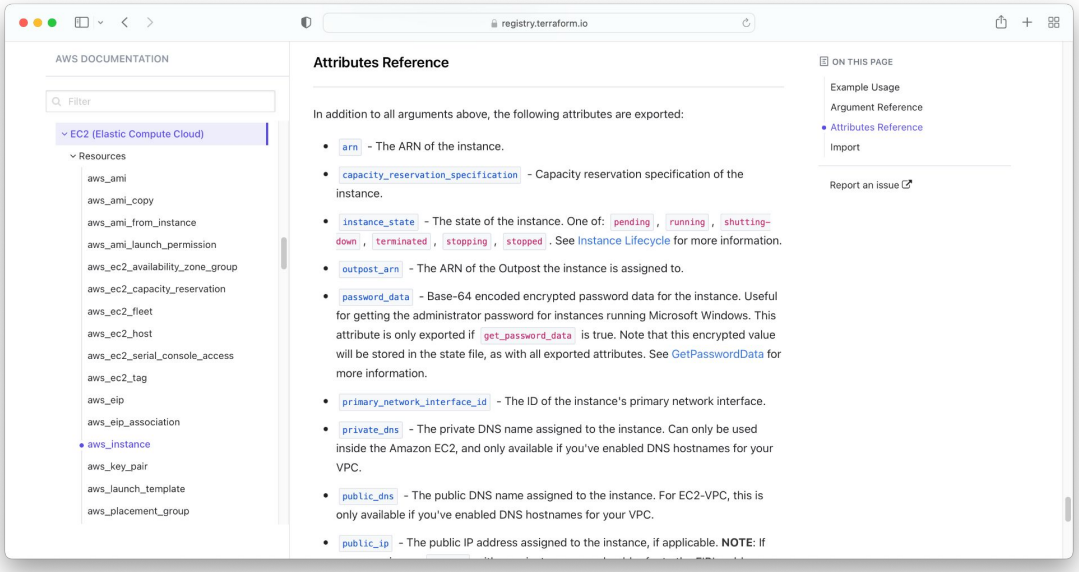
Dependency Lock File

Stores resolved versions + checksum for provider plugins (docs)

- Terraform creates or updates the dependency lock file each time you run the `init` command.
- The lock file is always named `.terraform.lock.hcl` and is meant to be checked into version control.
- Use the `-upgrade` flag with the `init` command to upgrade a provider plugin version.

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-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

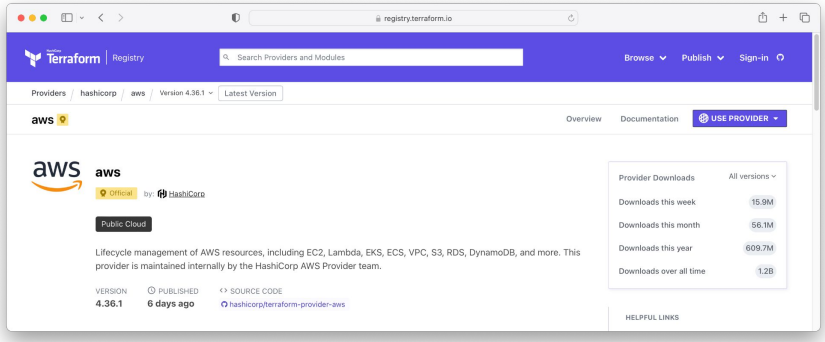
The free or paid Amazon Machine Image (AMI) to run on EC2 instance available via the EC2 Locator

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" { }
```



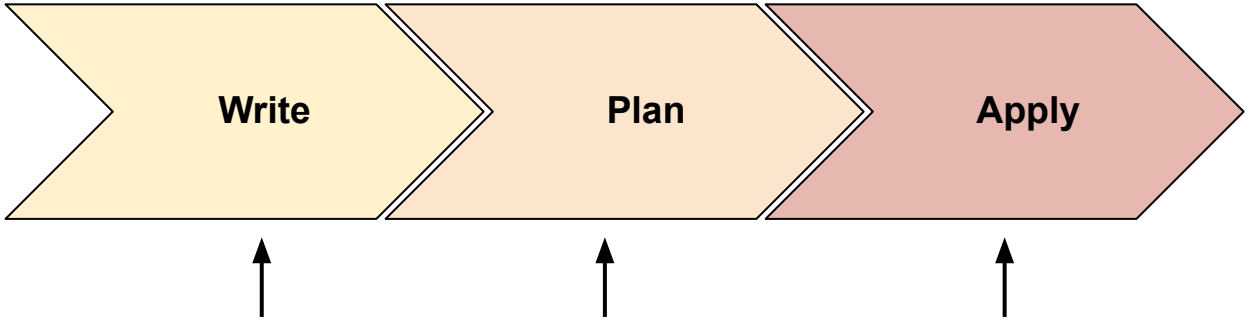
Use the prefix to determine provider on Terraform Registry and download latest version available

EXERCISE

Implementing a
Simple
Configuration File

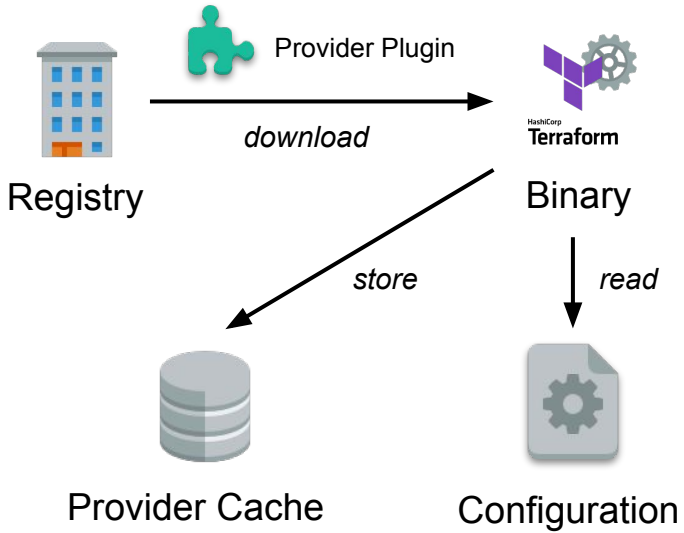
Terraform Core Workflow

Three-step approach on a high-level



Initialize Working Directory

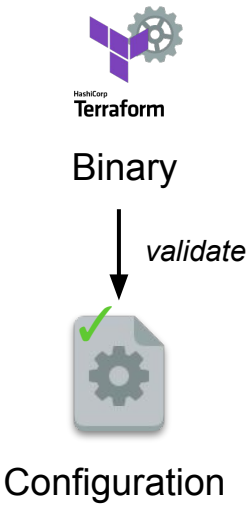
Prepare working directory, parse config, retrieve provider plugins




```
$ terraform init
```

Validate Configuration

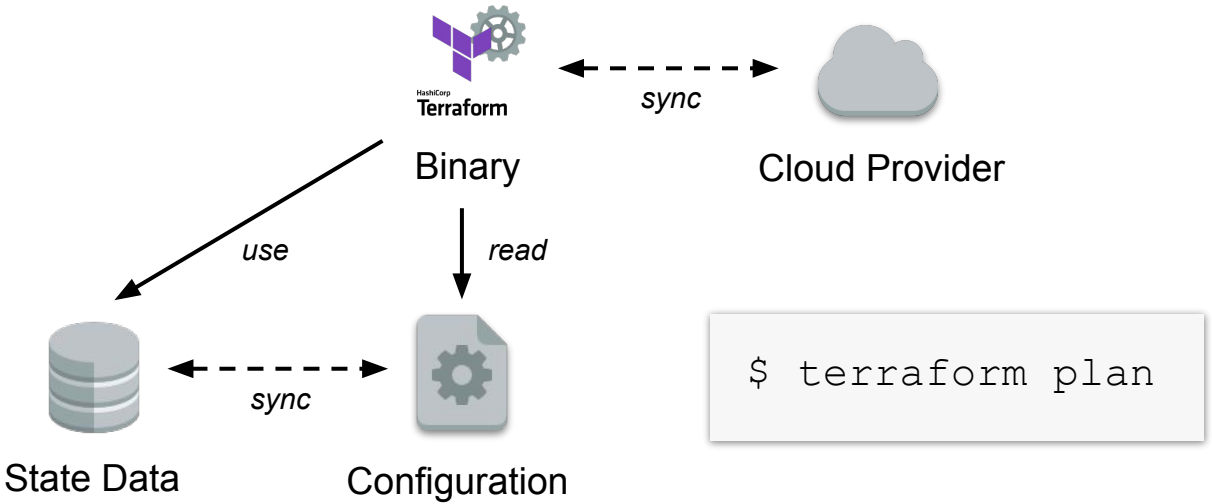
Check for syntax errors, doesn't guarantee successful deployment




```
$ terraform validate
```

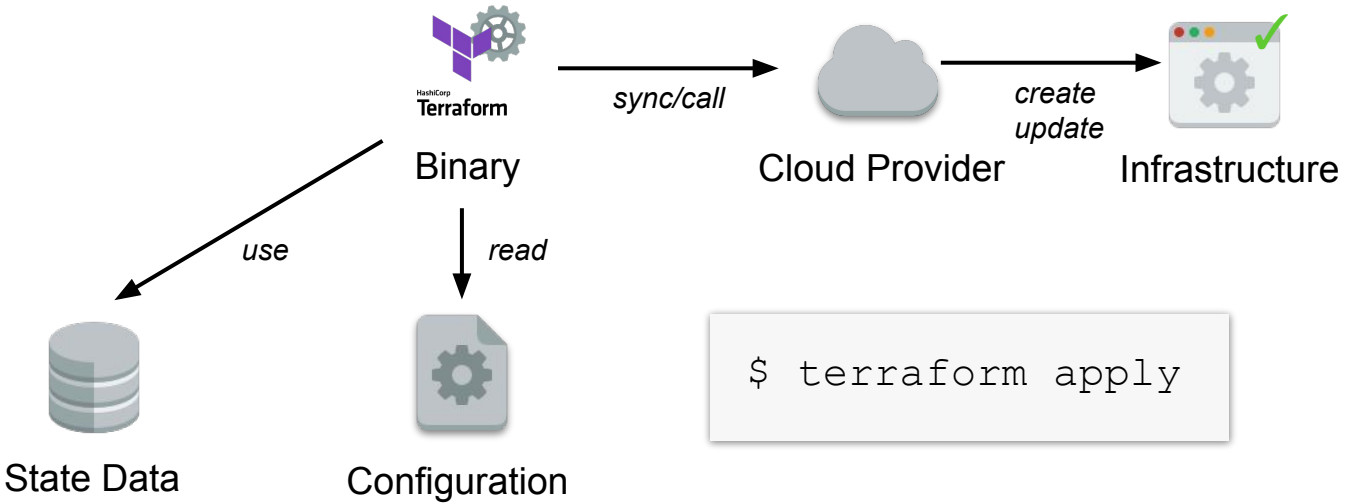
Plan Changes

Determine differences between local state and deployed infra



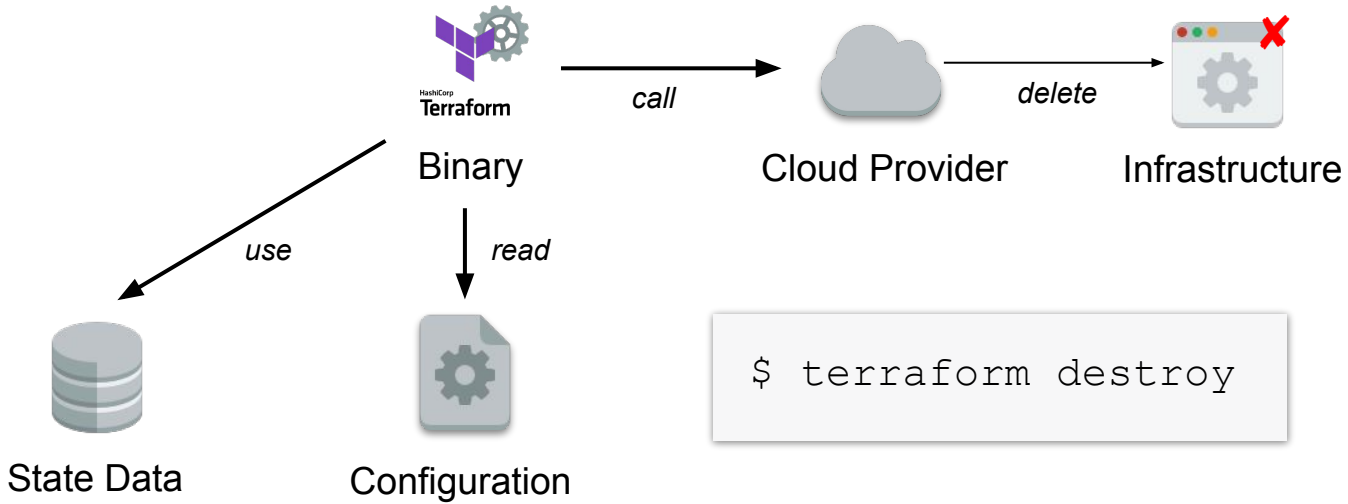
Apply Changes

Deploying the infrastructure or changing its delta



Destroy Infrastructure

Delete infrastructure in target environment based on state data



EXERCISE

Exercising the Core
Terraform Workflow

Q & A

Input Variables, Local/Output Values, and Data Sources

Data Types, Resource Addressing, Variables/Values

Data Types

A value follows one of the data types below

- Primitive
  - `string`
  - `number`
  - `bool`

- Collection or Complex
  - `list/tuple`
  - `set`
  - `map/object`

Primitive Data Types

Definition very similar to programming/scripting languages

```
# string
"hello" or "this is an example"

# number
25 or 4.632

# bool
true or false
```

Collection Data Type Examples

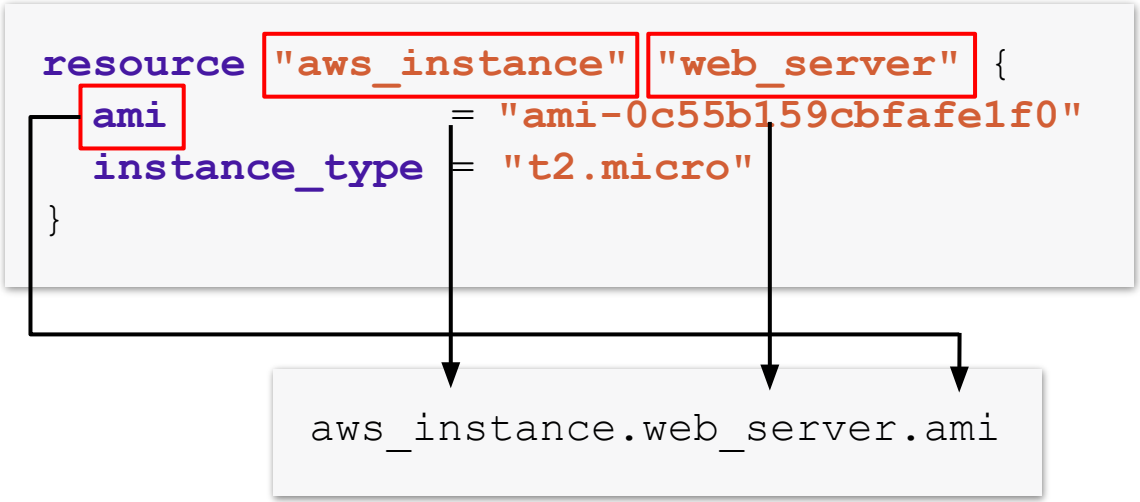
Items in collection can be defined on individual lines for readability

```
# list (or tuple)
["us-west-1a", "us-west-1c"]

# map (or object)
{name = "Mabel", age = 52}
```

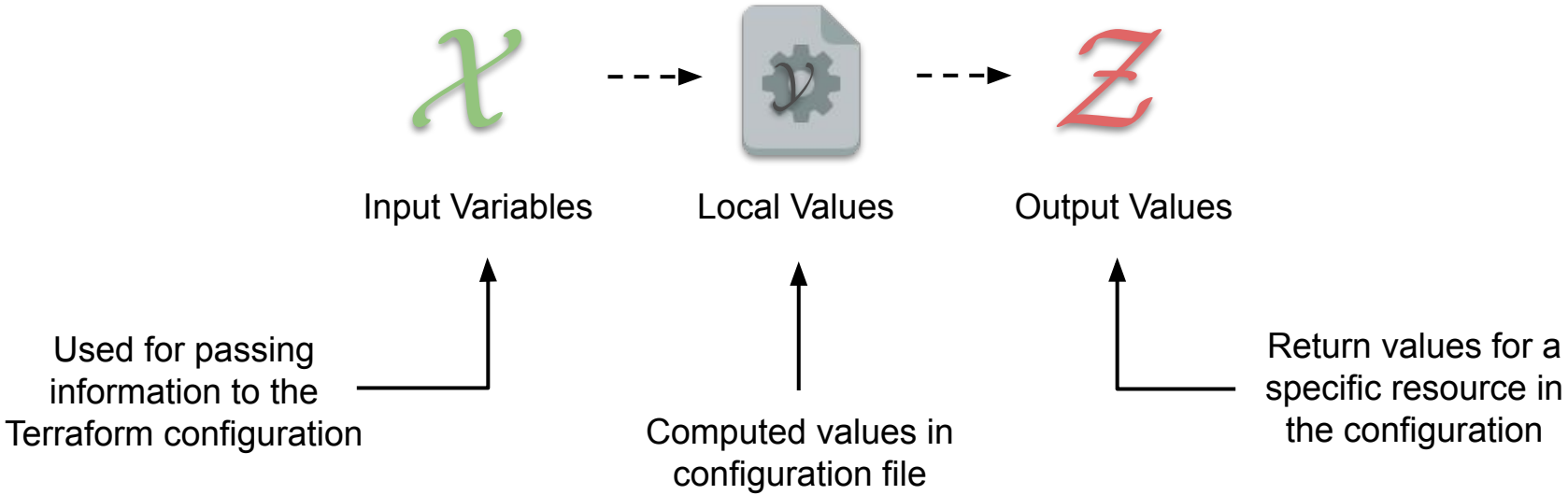
Resource Addressing

Referencing a variable or resource attribute in a different context



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, but not enforce

Input Variables

`variables.tf`

Local Values

`locals.tf`

Output Values

`outputs.tf`

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

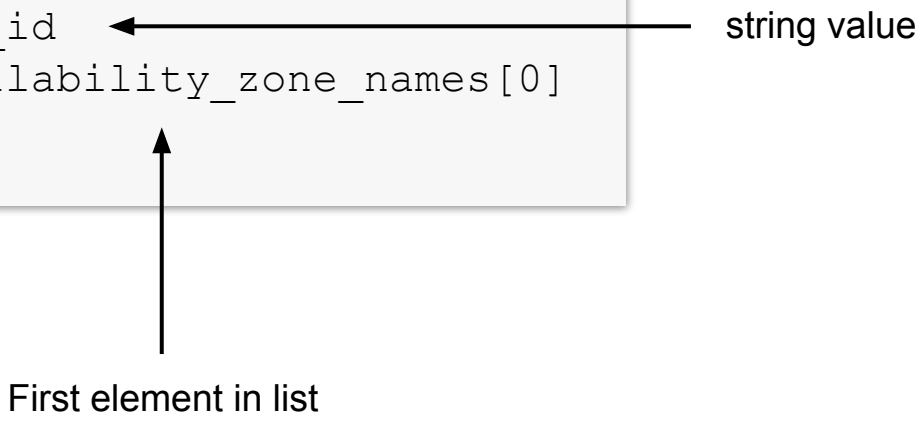
All arguments are optional

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 `null`

Referencing Input Variable Values

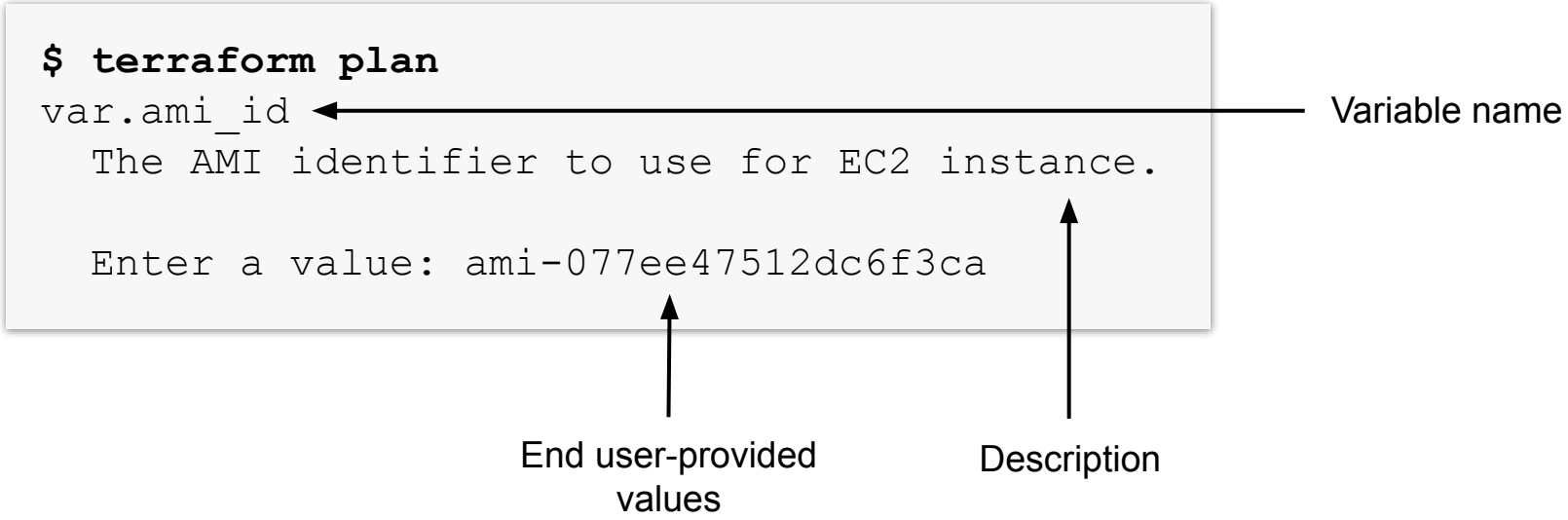
A variable can be used with the `var.name` notation

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



Requesting Variable Input Values

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



Variable Input Values from CLI

`-var` option when running the `plan` and `apply` commands

```
$ terraform apply -var="ami_id=ami-0c55b159cbfafe1f0"
...
$ 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-0c55b159cbfafe1f0"
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`

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-\"."
  }
}
```

Multiple Validation Rules

Reports all failures of the whole stack of validation blocks

```
variable "environment" {
  ...
  validation {
    condition = contains(["dev", "staging", "prod"], var.environment)
    error_message = "Environment must be dev, staging, or prod."
  }
  validation {
    condition = length(var.environment) >= 3
    error_message = "Environment name must be at least 3 characters."
  }
}
```

EXERCISE

Input Variable Definition and Consumption

Secrets in State

Terraform stores all values in state, including secrets

```
resource "aws_db_instance" "main" {
  password = var.db_password
}
```

- This is a major concern in enterprise Terraform
- State files contain passwords in plain text
- Even with S3 encryption at rest, anyone with state access can read secrets

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
```

Ephemeral Values

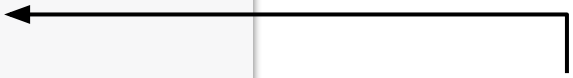
Never persisted to state or plan files

- **Ephemeral input variables:** Sensitive inputs not stored
- **Ephemeral resources:** Temporary data that exists only during operations
- **Write-only arguments:** Resource attributes not persisted (Terraform 1.11+)

Ephemeral Resources

Generate or fetch temporary data

```
ephemeral "random_password" "db" {
  length           = 16
  special          = true
  override_special = "!#$%&*()-_=+[]{}<>:?"
}
```



Executed during every `plan` and `apply`. Its values are not stored in state.

Ephemeral Resources Lifecycle

No data persistence of value

1. **Opening:** When required, the resource is "opened" (e.g., a secret fetched from a vault), generating a temporary value.

2. **Renewing (Optional):** If the resource's lease or validity is short, the provider can periodically "renew" it to extend its lifespan during the operation.

3. **Closing**: Once all dependent actions finish, the resource’s lease is revoked, ensuring cleanup and preventing data persistence.

Write-Only Arguments

Accept values but never store them (Terraform 1.10+)

```
resource "aws_db_instance" "main" {
  engine            = "postgres"
  instance_class    = "db.t3.micro"
  username          = "admin"

  password_wo         = ephemeral.random_password.db.result
  password_wo_version = 1
}
```

The `_wo_version` argument tracks when to update the value.

Write-Only Version Management

Triggers the update

- The version is just an integer - increment to trigger updates
- Some teams use a timestamp or hash instead
- Pattern for rotation: increment version in a PR, merge to trigger rotation
- Consider storing version in a separate variable for easier management

Supported Write-Only Resources

As of Terraform 1.12, write-only arguments are available in:

Provider	Resources
AWS	`aws_db_instance`, `aws_secretsmanager_secret_version`
Azure	`azurerm_mssql_server`, `azurerm_key_vault_secret`
Google	`google_sql_user`, `google_secret_manager_secret_version`

Check provider documentation for the latest supported resources.

EXERCISE

Ephemeral Value 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

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
```

Check Block

Validate infrastructure without blocking operations

```
check "website_health" {
  data "http" "app" {
    url = "https://${aws_lb.main.dns_name}/health"
  }

  assert {
    condition     = data.http.app.status_code == 200
    error_message = "Application health check failed."
  }
}
```

Failed checks produce warnings, not errors.

EXERCISE

Defining and Consuming a Data Source

Resource Dependencies

Implicit/explicit dependencies, Lifecycle block

Implicit Dependencies

Detects dependencies when one resource references another

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "web" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}
```

Creates an implicit dependency

Implicit Dependencies

Preferred approach - let Terraform figure it out

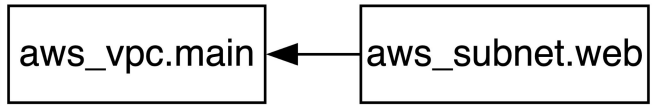
- Terraform builds a dependency graph and creates resources in the correct order

- The dependency graph is visible with `terraform graph` command

Rendered Graph

Provides a GraphViz representation

```
$ terraform graph
digraph G {
  rankdir = "RL";
  node [shape = rect, fontname = "sans-serif"];
  "aws_subnet.web" [label="aws_subnet.web"];
  "aws_vpc.main" [label="aws_vpc.main"];
  "aws_subnet.web" -> "aws_vpc.main";
}
```



https://dreampuf.github.io/GraphvizOnline/

When the approach doesn’t work

Sometimes Terraform can’t detect dependencies

- IAM policies that must exist before a resource can use them
- Security groups referenced by name instead of ID
- Resources that depend on side effects of other resources
- External systems that need time to propagate changes

Explicit Dependencies

Instance could launch before the IAM policy is attached

```
resource "aws_iam_role_policy" "s3_access" {
  name   = "s3-access"
  role   = aws_iam_role.app.id
  policy = jsonencode({...})
}

resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
  depends_on = [aws_iam_role_policy.s3_access]
}
```

Terraform cannot infer dependencies

When to use `depends_on`

Valid reasons for declaring explicit dependencies

- Hidden dependencies on IAM policies/roles
- Dependencies on resources accessed by name
- Cross-account or cross-region dependencies
- When order of creation matters for external reasons

When not to use `depends_on`

In most cases you won't need explicit dependency declarations

- When a direct reference creates an implicit dependency
- As a workaround for timing issues (use `time_sleep` instead)
- For data sources (can cause circular dependencies)

Lifecycle Block

Overrides Terraform's default resource management behavior

- The `lifecycle` block is one of Terraform's most powerful features
- It's a nested block that goes inside the `resource` block
- Available for all resource types - it's a meta-argument
- Customizes how Terraform manages the resource during create, update, and destroy operations

Core Lifecycle Arguments

Modify the default behavior of resource management

Argument	Runtime Behavior
`create_before_destroy`	Creates a new replacement resource before deleting the old one to ensure zero downtime.
`prevent_destroy`	Prevents Terraform from deleting a resource; any plan to destroy it will result in an error.
`ignore_changes`	Stops Terraform from reverting manual or external updates to specific resource attributes.
`replace_triggered_by`	Forces resource replacement when a specified dependency or attribute changes elsewhere.

`create_before_destroy`

Implementing a zero-downtime control

- **Default behavior:** Terraform destroys the old resource, then creates the new one.
- **Use case:** Zero-downtime deployments where the resource must remain available.
- With `create_before_destroy = true`: Terraform creates the new resource first, then destroys the old one.

create_before_destroy

New EC2 instance will be created before old one is destroyed

```
resource "aws_instance" "web" {
  ami             = var.ami_id
  instance_type = "t3.micro"

  lifecycle {
    create_before_destroy = true
  }
}
```

Usage Considerations

May require additional customization

- Resources with unique name requirements may conflict
- May require random suffixes or name prefixes
- Not all resources support having two instances simultaneously

Preconditions

Validate things outside the resource itself

```
resource "aws_instance" "web" {
  ...
  lifecycle {
    precondition {
      condition = data.aws_ami.selected.architecture == "x86_64"
      error_message = "The AMI must use x86_64 architecture."
    }
  }
}
```

Postconditions

Validate guarantees after Terraform creates a resource

```
resource "aws_instance" "web" {
  ...
  lifecycle {
    postcondition {
      condition = self.public_ip != ""
      error_message = "Instance must have a public IP address."
    }
  }
}
```

Preconditions vs Postconditions

Both block the operation on failure

Aspect	Precondition	Postcondition
When evaluated	Before resource creation	After resource creation
Can use `self`	No	Yes
Purpose	Validate assumptions	Validate guarantees
Typical use	Check inputs, data sources	Verify outputs, attributes

Q & A

Iteration Control, Functions, and Expressions

Loops, Dynamic Blocks, Built-In Functions

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-0c55b159cbfafe1f0"
  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

    ingress {
      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_name(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)
> 99

# base64-encode a string value
base64encode("Hello World")
> SGVsbG8gV29ybGQK

# extract a substring from a given string
substr("hello world", 1, 4)
> ello
```

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

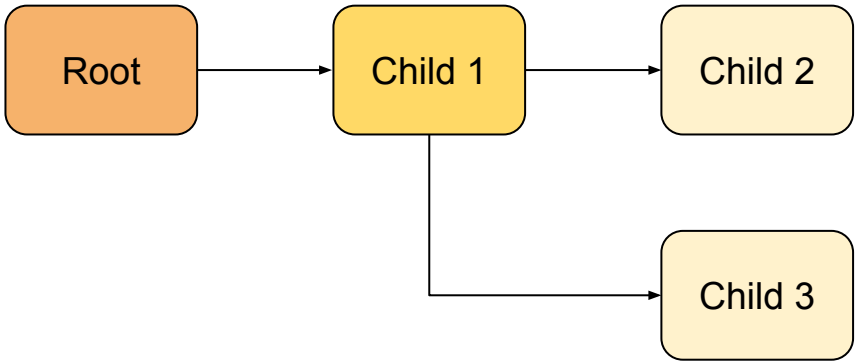
- A local path to a directory present on disk
- A registry like the Terraform registry
- A Git repository e.g. GitHub or BitBucket
- A HTTP(S) URL that follows a provider protocol

Root and Child Modules

Modules can be consumed on any level of the hierarchy

main.tf

```
module "child-1" {
  ...
}
```



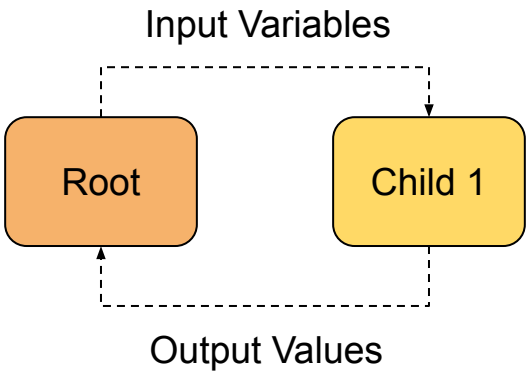
child-1/main.tf

```
module "child-2" {
  ...
}

module "child-3" {
  ...
}
```

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

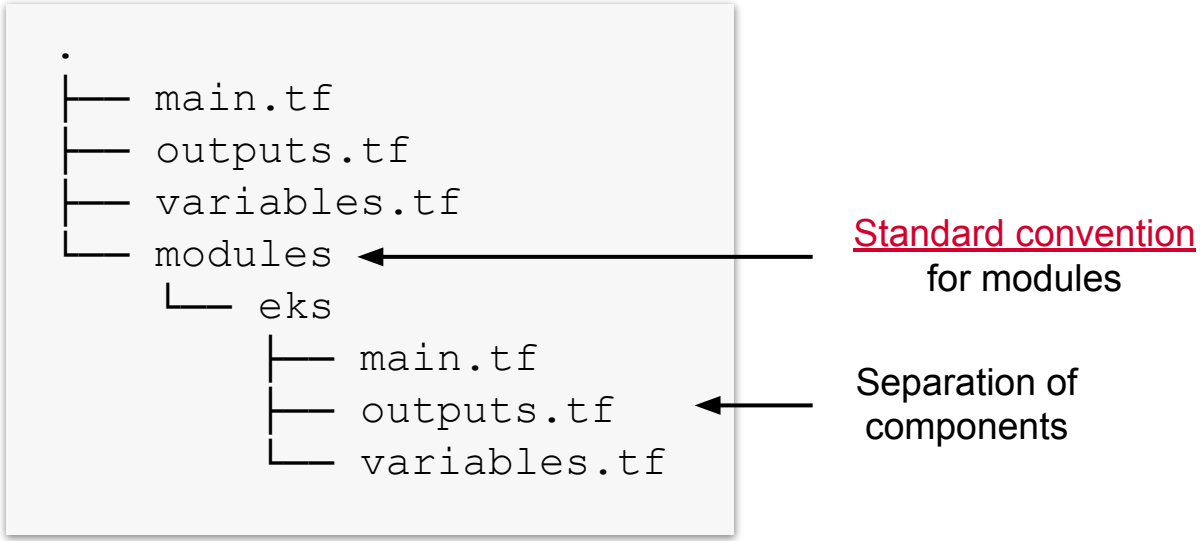
```
variable "cluster_name" {
  type = string
}

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
}
```

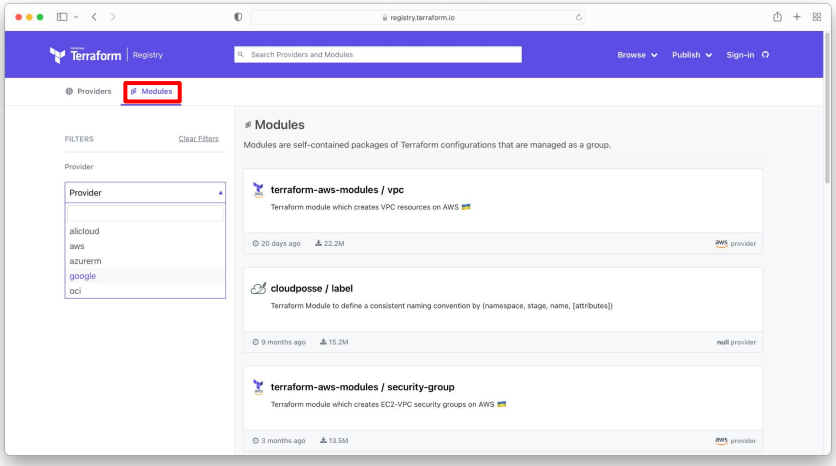
Location of module in directory structure

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



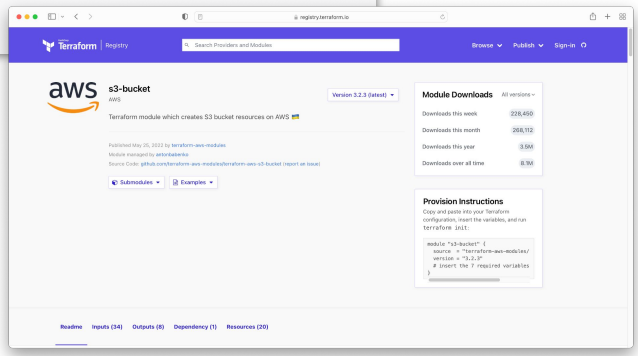
https://registry.terraform.io/browse/modules

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

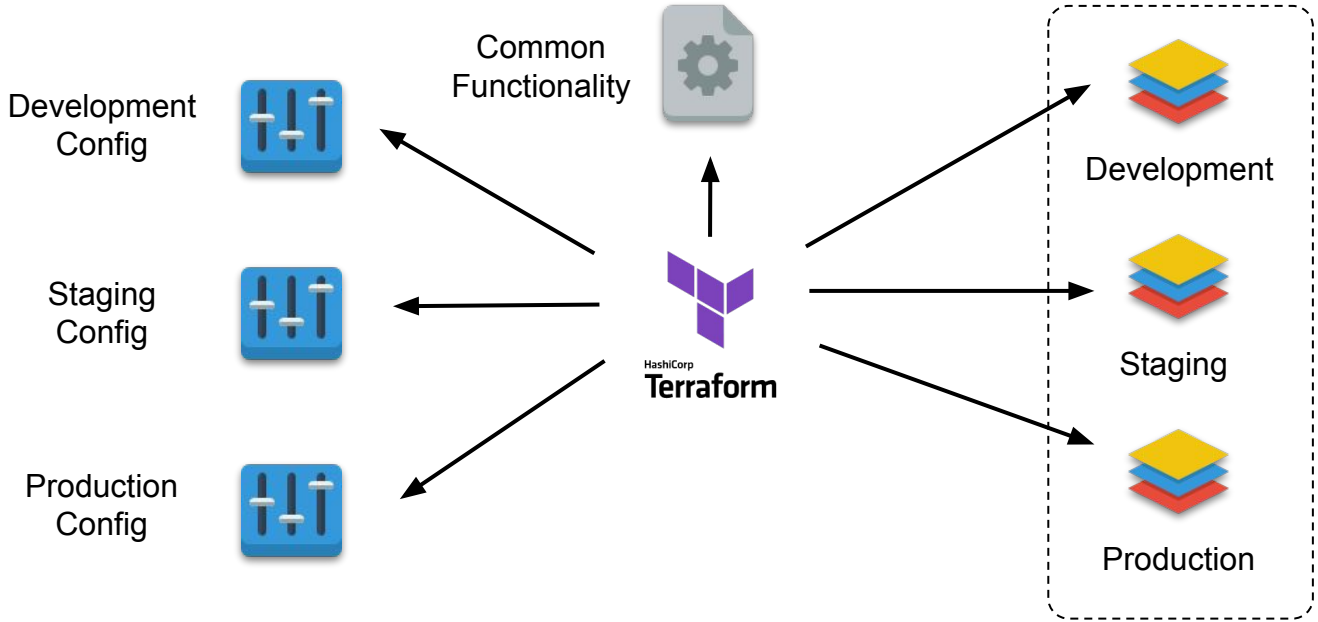
Q & A

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
- HashiCorp discourages use of workspaces and offers alternatives

The Default Workspace

Cannot be deleted and uses state file in root directory

```
$ terraform workspace show
default

$ terraform workspace list
* default
```

```
.
└── terraform.tfstate
```

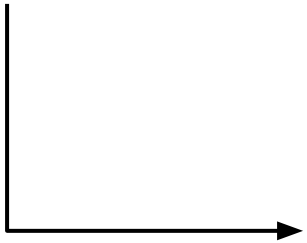
Marks the selected workspace with an asterisk character

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

Variable	Value
instance_type	t2.nano
instance_count	1

Staging

Variable	Value
instance_type	t2.micro
instance_count	2

Production

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-0c55b159cbfafe1f0"
  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

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

EXERCISE

Managing Multiple Environments With Workspaces

Implementing and Maintaining State

Local vs. Remote Backend, State Locking, Secret Management

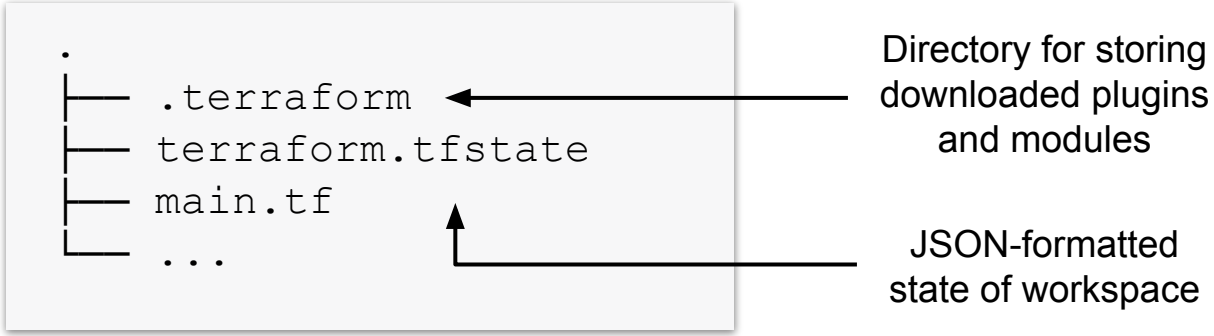
Holding State

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

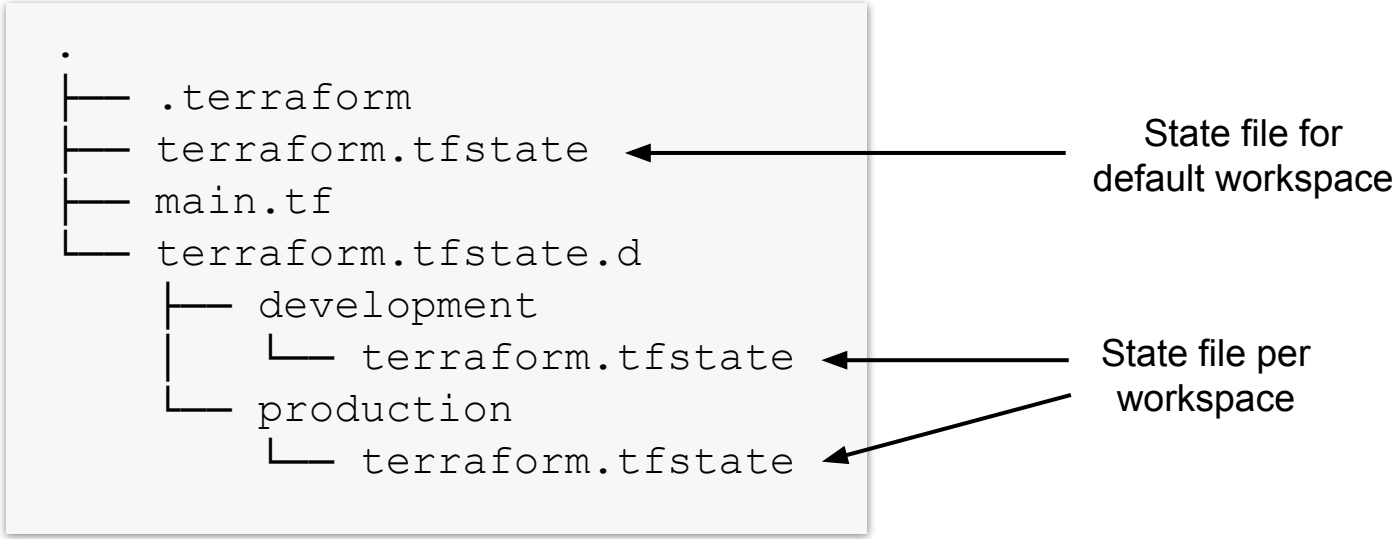
Default file & location can be changed by CLI or configuration

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

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

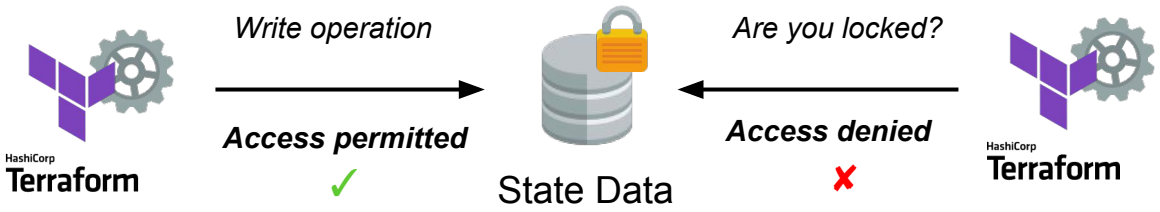
Local State File Per Workspace

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. HCP Terraform 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=organ
ization=company
```

HCP Terraform Configuration

Requires a specific notation (docs)

```
terraform {
  cloud {
    organization = "automated-ascent"
    hostname = "app.terraform.io"

    workspaces {
      tags = ["my-app-prod"]
    }
  }
}
```

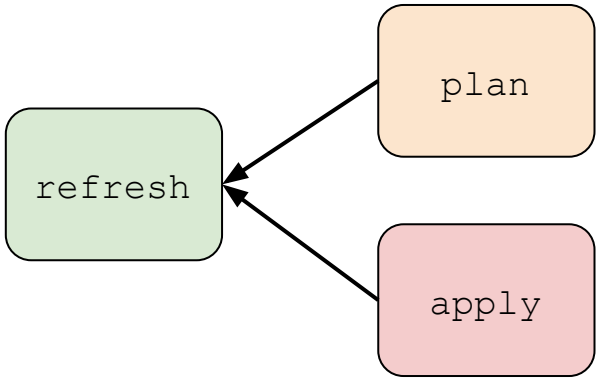
Secret Management & State

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




```
$ terraform refresh
```

No need to call explicitly

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

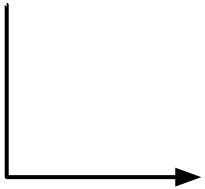
Check for syntactical errors in configuration files of a directory

- 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-0c55b159cbfafe1f0"
instance_type = "t2.micro"
```

```
$ terraform fmt
```

```
ami           = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
```

EXERCISE

Validating and
Formatting
Configuration Files

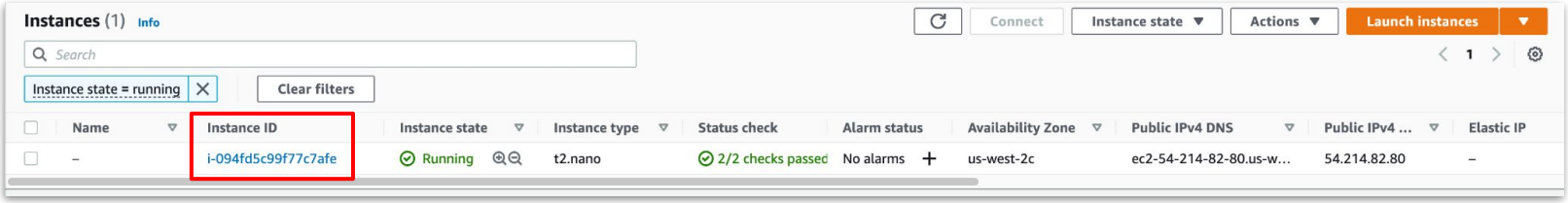
Importing Existing Infrastructure

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



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

Adding a Resource Config Block

The `import` command will attach state to configuration

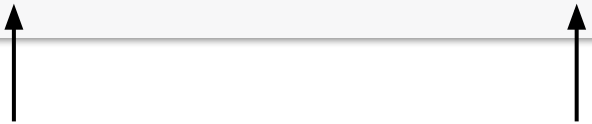
```
resource "aws_instance" "web_server" {}
```

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
```



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
```

Import Block

Terraform 1.5+ makes things easier (docs)

```
import {
  to = aws_instance.example
  id = "i-abcd1234"
}

resource "aws_instance" "example" {
  name = "backend"
}
```

```
$ terraform plan
```

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 output 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

Understanding HCP Terraform Capabilities

How Does it Help in an Enterprise Setting?

What is HCP Terraform?

Commercial, cloud-hosted distribution of Terraform

Product	Hosting	Use Case
Terraform (Open Source)	Your machine/CI	Individual use, custom pipelines
HCP Terraform	HashiCorp SaaS	Teams who can use cloud-hosted
Terraform Enterprise	Self-hosted (your infrastructure)	Air-gapped, regulated industries, data sovereignty

HCP Terraform Features

Collaboration and governance capabilities

Capability	Description
Remote State	Secure, versioned state storage with locking
Remote Execution	Runs happen in consistent cloud environment
Team Collaboration	Shared workspaces with access controls
VCS Integration	Automatic runs triggered by Git commits
Policy as Code	Enforce rules with Sentinel or OPA
Private Registry	Share modules and providers internally

Open Source Terraform vs HCP

Open Source for individual use, HCP is ideal for teams

Aspect	Open-Source Terraform	HCP Terraform
State storage	Local or self-managed backend	Managed, encrypted, versioned
Execution	Your machine or CI/CD	HashiCorp's infrastructure
Collaboration	Manual coordination	Built-in workspaces and teams
Secrets	You manage	Encrypted variables
Audit trail	None	Full run history
Cost	Free	Free tier + paid plans

Feature: Account Management

Authentication and authorization

- An 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 local Terraform workspace

- HCP Terraform and local Terraform uses the same terminology, however, the meaning for a *workspace* is different
- In HCP Terraform 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 and Open Policy Agent (OPA) are compliance frameworks 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

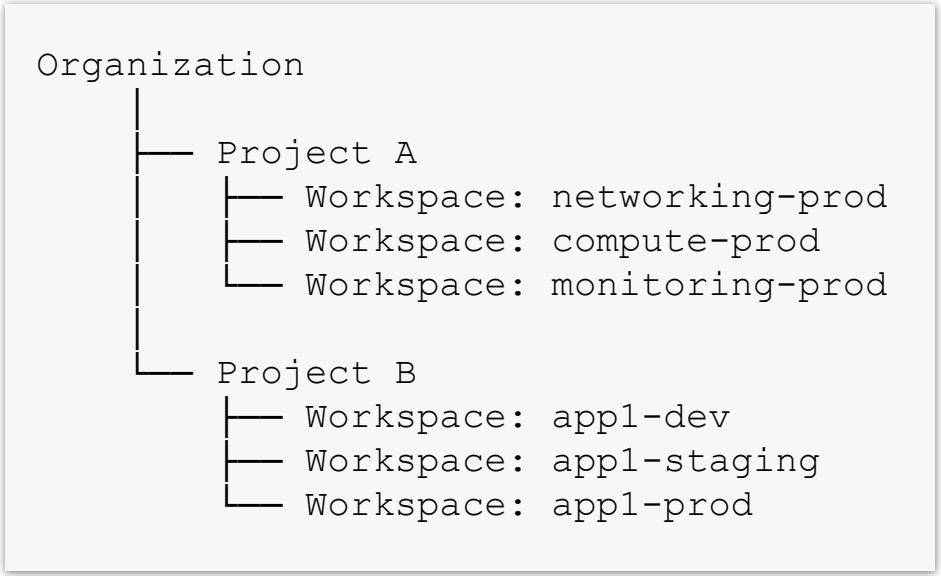
Other Notable Features

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

HCP Terraform Hierarchy

Organization(s) > Project(s) > Workspace(s)



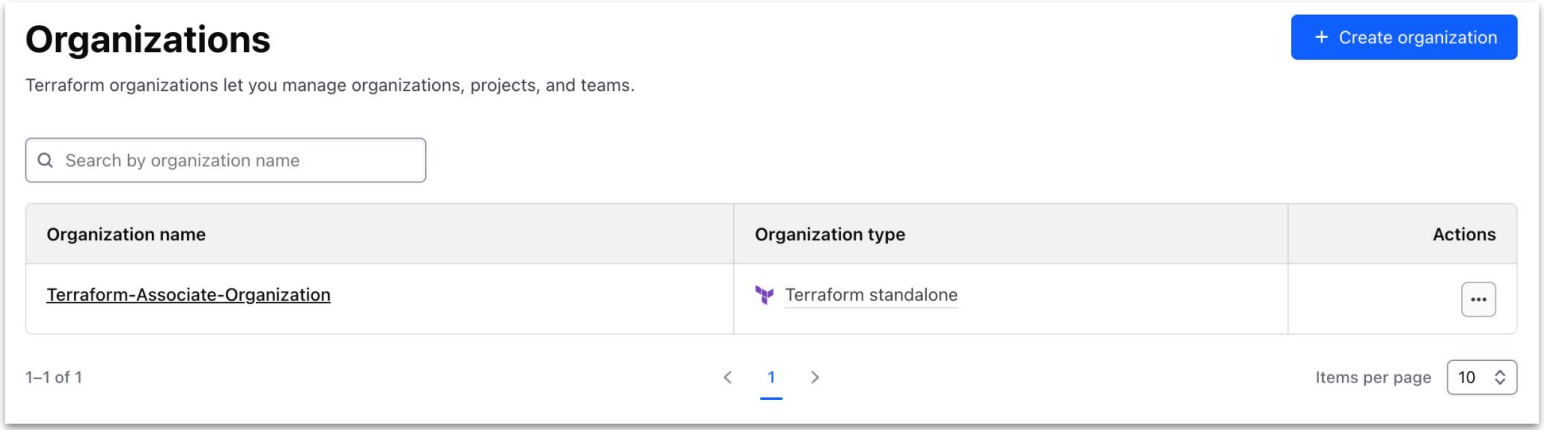
What are Organizations?

The root-level structure in HCP Terraform

- Most companies have ONE organization which maps to the company itself
- Some large enterprises have multiple (acquisitions, strict separation)
- Teams are defined at the org level, then granted access to projects/workspaces

Organization in UI

You'll be prompted to create a new organization to begin with



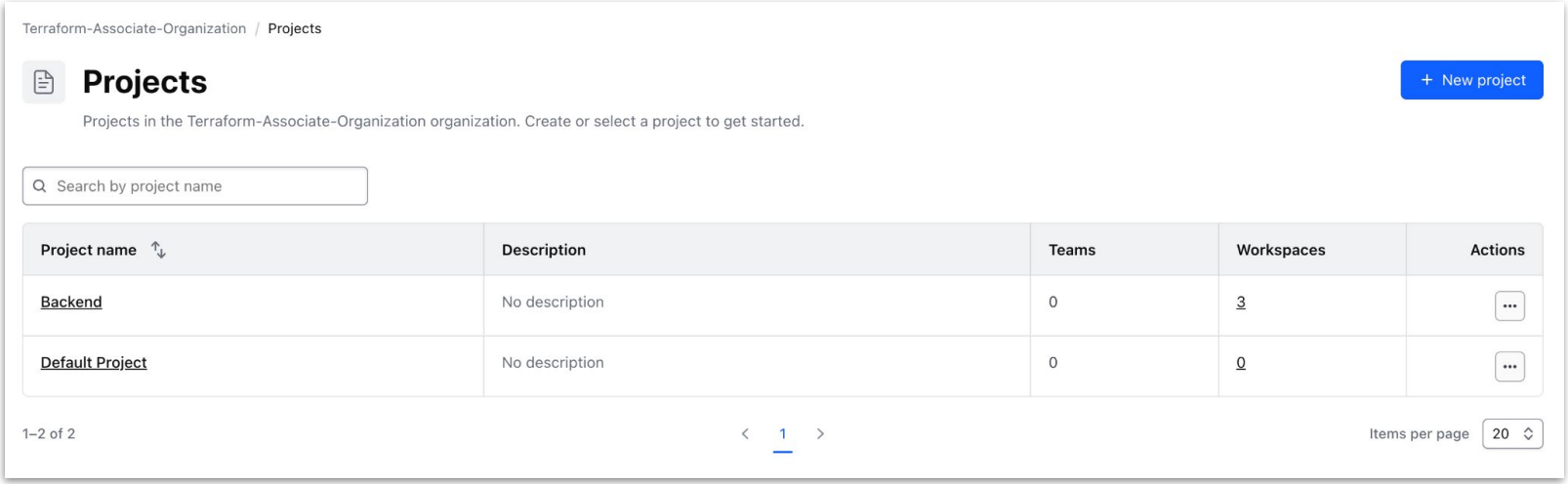
What are Projects?

Projects group related workspaces together

- Projects are a relatively new feature (2023)
- Before projects, organizations had flat lists of workspaces
- Projects solve the "too many workspaces" problem
- Think of them as folders for workspaces

Projects in UI

To be created underneath an Organization



Project Benefits

It's highly recommended to use projects

- **Focused View**
  - See only relevant workspaces
  - Filter by project in UI

- **Simplified Management**
  - Apply variable sets to all workspaces in a project
  - Assign team permissions at project level
  - Consistent policy enforcement

Project Benefits

It’s highly recommended to use projects

- **Least Privilege**
  - Teams access only their projects
  - No organization-wide permissions needed

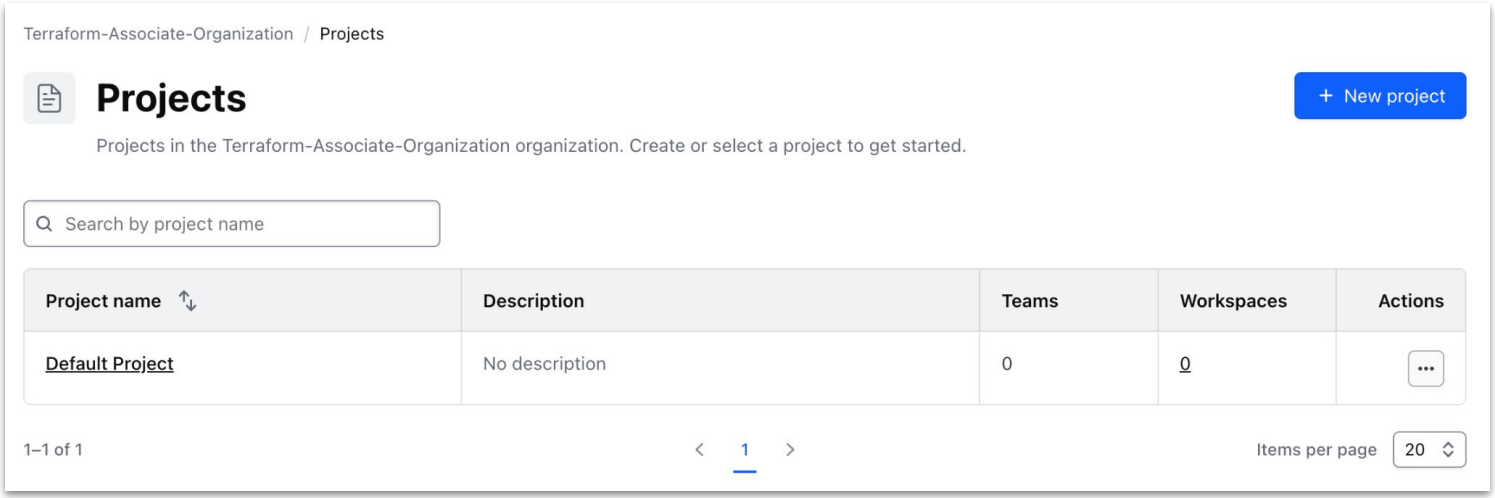
The Default Project

Every organization has a default project

- Cannot be deleted (can be renamed)
- New workspaces go here by default
- Catch-all for unassigned workspaces
- Best practice: Create specific projects and move workspaces out of “Default”

Default Project in UI

Automatically available with creation of an Organization



Project Permissions

Teams can have different permissions per project

Permission	Capabilities	Use Case
Read	View workspaces and runs	View-only access, good for auditors
Plan	Queue plans	Can test changes but not apply - good for junior devs
Write	Queue applies	Full operational access
Admin	Manage workspace settings	Can change settings, manage variables

Project-Level Variable Sets

“tfe provider” interacts with resources supp. by HCP Terraform

```
resource "tfe_variable_set" "aws_creds" {
  name         = "AWS Credentials"
  organization = "my-org"
}

resource "tfe_project_variable_set" "platform_aws" {
  project_id      = tfe_project.platform.id
  variable_set_id = tfe_variable_set.aws_creds.id
}
```

What are Workspaces?

A workspace is an isolated environment containing:

- **Terraform configuration:** Your infrastructure code
- **State file:** Current state of managed resources
- **Variables:** Input values for the configuration
- **Run history:** Record of all plan/apply operations
- **Access controls:** Team permissions

Local Directories vs Workspaces

Key benefits: collaboration, audit trail, access control

Local Terraform	HCP Terraform
Separate directories	Separate workspaces
Local state files	Remote state storage
Manual collaboration	Built-in collaboration
No run history	Full audit trail
No access control	Full audit trail

Workspace Naming Conventions

Currently not enforced

Recommended format:

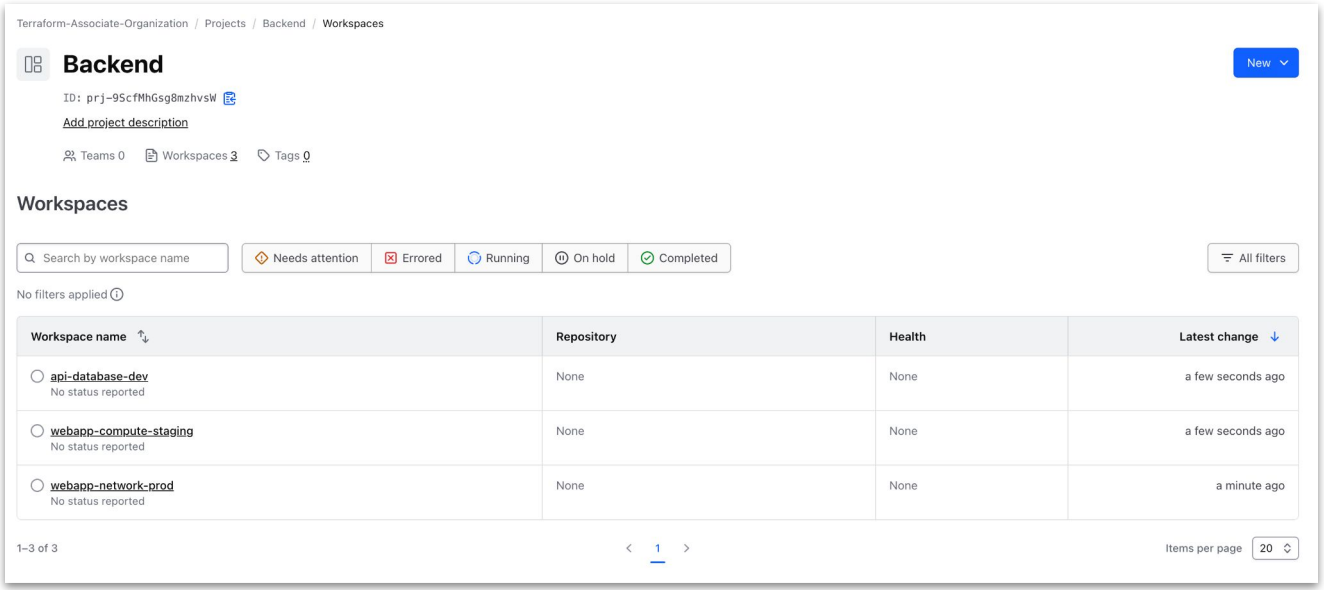
```
<application>-<layer>-<environment>
```

Examples:

- `webapp-network-prod`
- `webapp-compute-staging`
- `api-database-dev`

Workspaces in UI

Created with New button available to a Project



Q & A

EXERCISE

Apache Web Server
on AWS Accessible
via the Internet

Summary & Wrap Up

Last words of advice...

O'REILLY®

Thank you