# HashiCorp Certified: Terraform Associate (002)

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

The Terraform Associate certification is for Cloud Engineers specializing in operations, IT, or development who know the basic concepts and skills associated with open source HashiCorp Terraform. Candidates will be best prepared for this exam if they have professional experience using Terraform in production, but performing the exam objectives in a personal demo environment may also be sufficient. This person understands which enterprise features exist and what can and cannot be done using the open source offering. Visit our exam partner to [schedule and take the exam](https://hashicorp-certifications.zendesk.com/hc/en-us/articles/360049382552).

**Prerequisites**

* Basic terminal skills
* Basic understanding of on premises and cloud architecture

**Product Version Tested**

Terraform 1.0 and higher.

**Preparing for the Exam**

Certification preparation learning guides for the new version of the Terraform Associate will be posted soon. For now, use the current [Terraform Associate Tutorial List](https://learn.hashicorp.com/collections/terraform/certification-associate-tutorials) to start your studying.

**Exam Details**

|  |  |
| --- | --- |
| **Assessment Type** | Multiple choice |
| **Format** | Online proctored |
| **Duration** | 1 hour |
| **Price** | $70.50 USD plus locally applicable taxes and fees Free retake **not included** |
| **Language** | English |
| **Expiration** | 2 years |

# Exam Objectives

## **Understand infrastructure as code (IaC) concepts**

## Explain what IaC is

* Human readable files that defince infrastructure for both cloud and on-prem
* Immutable
* declarative
* Providers Utilizes APIs
* Allows scalability to be automated
* Workflow stages (same on any platform)
  + Write
    - Define resources
    - Can be across multiple cloud providers and services
  + Plan
    - Terraform creates an execution plan describing infrastructure it will
      * Create
      * Update
      * Destroy
  + Apply
    - On approval, terraform performs proposed operations in correct order
* Types of IaC
  + AdHoc Scripts
    - Better than nothing still not framework
  + Configuration Management Tools
    - Chef
    - Puppet
    - Ansible
    - SaltStack
    - Install and manage software o existing servers
    - Usually mutable
      * Each server slowly has config creep
  + Server Templating tools
    - Docker
    - Packer
    - Vagrant
    - Full server template including any software
  + Orchestration tools
  + Provisioning Tools
    - Terraform
    - CloudFormation
    - OpenStack
    - Create the servers (infrastructure) themselves
* Procedural vs Declarative
  + Procedural
    - Code defines step by step process
    - When you run the code a second time it creates new resources
    - Does not track state
  + Declarative
    - Desired state
    - To add just add more resources to the code and it will add only new, not change existing
    - Immutable
    - Code always represents the current state

### Describe advantages of IaC patterns

* Allows you to use only what you need instead of keeping all the time
* Immutable
* Versioning
* Reuse components
* Self service
* Configuration consistency
  + Idempotent
  + Consistent
  + Repeatable
  + Predictable
* Minimize risk
  + Automates process
  + Form of documentation
* Increased efficiency in dev
  + Sandbox environment same for all and built quickly
* Reduce cost
  + Tear down infrastructure when not needed
  + elastic

## **Understand the purpose of Terraform (vs other IaC)**

https://developer.hashicorp.com/terraform/intro/vs

## Explain multi-cloud and provider-agnostic benefits

* What is Terraform
  + Open source
  + Masterless
    - No server to hold configs
  + Agentless
  + Used to do this to infrastructure
    - Build
    - Change
    - Manage
  + Used files ending in
    - .tf
    - .tf.json
  + Same workflow
* Multi-Cloud
  + Single workflow
  + Works on public and private clouds
  + Cloud agnostic
* Infrastructure defined as code
  + Shared and reused
  + Modulare
* Execution plan
  + Terraform plan
  + Information about what terraform wil do
  + Shows what will change
* Resource Graph
  + Efficient build
  + Parallel execution of anything not dependent
* Comparison
  + CloudFormation
    - Not Cross Platform only for AWS
    - Uses YAML or JSON
    - Good Conditional functions
    - Wait condition and creation policy
    - Not quite as modular (nested stacks)
    - Not as good validation
    - Harder to read than terraform
    - State file stored native to AWS
  + Azure ARM
    - Only Azure
    - JSON
    - Not as modular
    - Not as good validation
    - Harder to read JSON
    - State stored in Azure
  + Google Cloud Deployment Manager
    - Only GCP
    - YAML, Jinja or Python
    - Not as modular
* Terraform Architecture
  + Terraform Core
    - Binary written in GO
    - Communicates with terraform plugins via RPC (remote procedure calls)
    - Terraform CLI
    - Responsibilities
      * IAC: reading and interpolating configuration files and modules
      * Resource state management
      * Resource graph construction
      * Plan execution
      * Communication with plugins
  + Terraform Plugins
    - Exposes an implementation for a specific service ( AWS, Azure, VMWare etc) or provisioner (BASH)
    - Defined in the terraform configuration file
    - Responsibilities Provider Plugin
      * Init of any included libraries
      * Auth with infra provider
      * Def resources map to specific services
    - Responsibilities Provisioner Plugin
      * Execute commands on resource following creation or destruction
    - Terraform init looks for plugins in default locations
      * Can be overridden with -plugin-dir=<PATH>
    - Upgrading Plugins
      * -upgrade checks found version with hashicorp for newer version

## Explain the benefits of state

* State files are stored on the local file system or remote backend ( think S3 )
* Declarative
* Terraform config always shows the current state
* Maps configurations to real world
  + One remote object mapped to one resource instance
* Stores metadata
  + Dependencies
* Stores cache of all resource values in state
  + For performance
  + State is available in the file so terraform does not have to query the provider.
* Everyone should be working with the same state file (
  + Store it on a shared location
  + Remote locking
    - Only one person can run terraform against state file at a time.
* State keeps track of resources already provisioned and only modifies what needs to be changed
* Backends
  + Local backend
    - Default stored on local directory where terraform config is kept.
    - Workspaces
      * A unique state in the same directory.
  + Remote backend
    - Allows working in teams. Everyone uses same state file
    - Sensitive data stays off local disk
    - Remote operations – some commands can be run remotely so they won’t depend on your system being logged in after you run terraform apply
    - Classes of backends
      * Standard
        + Includes state management and possibly locking
      * Enhanced
        + Includes remote operations on top o standard features
        + Terraform cloud or enterprise
    - Configured in either a terraform / backend block or a backend file
      * $.backendname.tfbackend

## **Understand Terraform basics**

### Install and version Terraform providers

* Provider
  + Plugin downloaded when you run Terraform Init
  + Provides the code so terraform knows what to do to interact with APIs and resources
  + Can be defined in any .tf or tf.json file
    - BEST PRACTICE
      * Create providers.tf or required\_providers.tf
  + Can also specify version
    - Or range of versions for the provider
  + Required\_Provider block
    - Sets the provider version for all nested configs
    - Can have multiple provider blocks for each provider
    - Can reference variables but not resource attributes
  + Alias:
    - Multiple configurations for the same provider
    - Muttiple regions of a cloud provider
    - Configuration\_alias in required\_providers defines alias
    - Procider without the configuration\_alias is considered default.
      * If all have alias then implied empty provider is default
* Provisioner
  + Terraform cannot track state of these as they are usually scripts
  + Recommended as last resort
  + Use case
    - Loading data into a VM
    - Bootstrappin a VM for a config manager
    - Saving data locally on your system
  + Provisioner types
    - Local-exec
      * Runs code locally to extract some information.
      * Can also define what to do when this local code fails
        + On\_failure
      * What to do during destroy
        + When = destroy
        + Destroy provisioners are run before resorce is destroyed
        + If fails will rerun when you run terraform apply
      * Can have multiple local-exec in same resource block
    - File
      * Copies files or directories from the machine where terraform is running to the new resource.
      * Ssh or winrm connection
    - Remote-exec
      * Run script on remote resource after it is created
      * SSH or WINRM
      * Inline
        + Commands executed in sequence
* Resources
  + Part of configuration
  + Block create, update, destroy actual resources
  + Use plug-in to do the actual work
  + Lifecycle setting
    - Tell terraform how to create / update / delete
* Variables
  + Input Variable
    - Defined in main.tf or variables.tv
    - Defined but no value and during runtime you will be queried for input
  + Can reference resources by what I call pseudo vars
    - <Resource Name>.<Attribute>
  + Environment Variables
    - Begin with TF\_VAR\_
    - Export TV\_VAR\_rgname=example-rg
  + Defined in file
    - Terraform.tfvars
    - Terraform.tfvars.json
    - Or ending in .auto.tfvars or .auto.tfvars.json
    - Any other file name and it will need to be called out in the terraform execution command
* Output
  + Return values of a Terraform resource / module /data
  + Use case
    - The output of one resource etc can be used in another if there is a dependency
    - Print output to CLI by running Terraform Apply
    - With remote state, other configs can help with root module outputs
  + Helps you expose some info from resource blocks
  + Optional arguments
    - Description
      * Describes the output
    - Sensitive
      * Prevents output from showing on the CLI via Terraform apply
      * Could still be seen via other ways
    - Depends\_on
* Data Sources
  + Allow you to extract output or info from already existing resources provisione by any other terraform config or manually or by any other means.

### Describe plugin-based architecture

* [Plugin Development - How Terraform Works With Plugins | Terraform | HashiCorp Developer](https://developer.hashicorp.com/terraform/plugin/how-terraform-works)
* Terraform Core
  + Binary written in GO
  + Communicates with terraform plugins via RPC (remote procedure calls)
  + Terraform CLI
  + Responsibilities
    - IAC: reading and interpolating configuration files and modules
    - Resource state management
    - Resource graph construction
    - Plan execution
    - Communication with plugins
* Terraform Plugins
  + Exposes an implementation for a specific service ( AWS, Azure, VMWare etc) or provisioner (BASH)
  + Defined in the terraform configuration file
  + Responsibilities Provider Plugin
    - Init of any included libraries
    - Auth with infra provider
    - Def resources map to specific services
  + Responsibilities Provisioner Plugin
    - Execute commands on resource following creation or destruction
  + Terraform init looks for plugins in default locations
    - Can be overridden with -plugin-dir=<PATH>
  + Upgrading Plugins
    - -upgrade checks found version with hashicorp for newer version

### Write Terraform configuration using multiple providers

### Describe how Terraform finds and fetches providers

## **Use Terraform outside of core workflow**

### Describe when to use terraform import to import existing infrastructure into your Terraform state

* If you create infrastructure outside of terraform, you must create a configuration matching this and then import using terraform import

### Use terraform state to view Terraform state

* Terraform\_remote\_state
  + Use to retrive output from state another state file

### Describe when to enable verbose logging and what the outcome/value is

## **Interact with Terraform modules**

### Contrast and use different module source options including the public Terraform Module Registry

### Interact with module inputs and outputs

### Describe variable scope within modules/child modules

### Set module version

## **Use the core Terraform workflow**

### Describe Terraform workflow ( Write -> Plan -> Create )

* Loops
  + Count
    - Iterates resource x number of times
    - Can use Count.index to make unique names
    - Can use length of list to tell count how many
    - Limitations
      * Can’t use nested counts
      * When modify resource that has a count, terraform treats count as an array, and as it is change it will need to destroy and recreate
  + For\_each
    - Loop over set of strings or maps
      * Can convert list to set using function: toset()
    - Can create multiple copies of entire resource
    - Multiple copies of inline code
    - For\_each returns a map of resources instead of a list so removing a middle resource does not cause destroy and recreate
  + For
    - Use for both lists and maps
* Debugging
  + TF\_LOG
    - Environment variable to enable detailed logging
    - Trace
      * Default
      * Most detailed
    - Debug
    - Info
    - Warn
    - Error
  + TF\_LOG\_PATH
    - Environment variable where to save logs
  + Crash.log
    - Log file create if terraform crashes
      * Includes crash log
        + Not useful to us on devs
      * Panic message and backtrace

### Initialize a Terraform working directory (terraform init)

* Downloads the required providers

### Validate a Terraform configuration (terraform validate)

### Generate and review an execution plan for Terraform (terraform plan)

* Shows what terraform wil do before actually making any changes

### Execute changes to infrastructure with Terraform (terraform apply)

* Actually performs any changes specified by the terraform files

### Destroy Terraform managed infrastructure (terraform destroy)

### Apply formatting and style adjustments to a configuration (terraform fmt)

## **Implement and maintain state**

### Describe default local backend

* Terraform.tfstate
* Located in the directory you run terraform init from
* Json file
* Terraform plan is a diff between running resources and state file
* Terraform import
  + Imports into state
* Terraform state
* Do not store state files in version control
  + Easy to forget to pull down latest changes
  + Do not provide locking
  + All data stored in plain text
    - Secrets could be included
* Doesn’t work well with teams. Need to use shared storage

### Describe state locking

### Handle backend and cloud integration authentication methods

### Differentiate remote state back end options

* Shared storage states
* Allows teamwork
* Supported backends
  + AWS S3
  + Azure storage
  + Google cloud storage
  + Terraform cloud
  + Terraform pro
  + Terraform enterprise
* Provides shared storage
* Provide locking
  + AWS requires DynamoDB for locking
* Provide encryption
* Add to terraform block to poin state to remote backend
  + Backend = {}
  + Rerun terraform init
* Terraform automatically pulls and pushes state from / to this location
* Limitations
  + Must create the shared location prior to configuring terraform to use it.
  + Backend block does not allow use of variable
    - Remote info must be copied into each modules terraform config
    - Must not copy key. This must be unique for each module otherwise you will overwrite an existing state
    - Can use partial config and then pass repeated info via terraform init -backend-config=…
* Isolation
  + You should not have the same file for all environments
  + Isolation by workspaces
    - Quick, isolated tests on same config
    - Separate named workspaces
    - Terraform workspace to switch between workspaces
    - Can reference workspace in config and modify depending on the workspace
    - Drawbacks
      * Not very isolated
        + Uses same key
        + Depends on folder structure
      * not visible in the code or terminal
        + difficult to maintain
      * error prone
  + Isolation via file layout
    - Production use cases with strong separation between environments
    - Puts each environment in separate folder
    - Configure different backend for each environment

### Manage resource drift and Terraform state

### Describe backend block and cloud integration in configuration

### Understand secret management in state files

* Should store secrets in config
* No matter how you deal with storing passwords, they are always stored in state file in plain text
* Terraform data source
  + AWS secrets manager and secret version
  + Aws systems manager parameter store
  + Aws key management service
  + Google cloud KMS
  + Azure key vault
  + Hashicorp vailt
* Manage outside of terraform
  + Lastpass etc and pass the password via environment variable

## **Read, generate, and modify configuration**

### Demonstrate use of variables and outputs

### Describe secure secret injection best practice

### Understand the use of collection and structural types

### Create and differentiate resource and data configuration

### Use resource addressing and resource parameters to connect resources together

### Use HCL and Terraform functions to write configuration

* Functions
  + Convert data
  + Terraform Console
    - Allows you to test terraform functions
    - Does not change the stat file

### Describe built-in dependency management (order of execution based)

## **Understand Terraform Cloud capabilities**

## Explain how Terraform Cloud helps to manage infrastructure

## Describe how Terraform Cloud enables collaboration and governance