Introduction to Terraform

Terraform

Agenda

- Infra as Code
- Terraform

Infra as Code

Terraform

What is Infra As Code (IaC)?

- Infrastructure as code is the approach to defining your infrastructure through source code that can then be treated just like any software system
- Infrastructure can be computing (like VMs), networking, security, any cloud managed service and resource (like Kubernetes clusters, serverless, etc.), GitHub repos, etc.
- This code (as any type of code) must be kept in source control to allow auditability, versioning all full integration with CI/CD
- Natural practice with cloud computing but can be use on several on-prem virtual environments

IaC: Benefits

- Faster and easier way to provisioning, validate and reconfigure your infra
- Help on configuration drift (consistency)
- Control cost on dynamic environments
- Full integration with source control
- Versioned together with source code and pipelines

IaC: Benefits

- Serves as infrastructure live documentation using declarative configuration
- Easy and recommended integration with CI/CD process, adding additional layer of security
- Allow you to test your infra definition
- Allow shift-left on security analysis on your infra

IaC: Declarative configuration

- Declarative configuration allow to define desired state on a more human-readable style
- You define what you want to achieve at the end
- How to implement your configuration is not your concern. Let the tooling do that for you
- Opposite of imperative configuration like scripting where you need to define all the steps
- Your configuration is idempotent, means you may ask to get your desired state as much you need and at the end you get always the same outcome
- With imperative configuration you may get the same but you need to do it by yourself

IaC vs Configuration Management

- Infrastructure as code defines all your infra resources (computing, networking, managed services)
- Configuration management defines configuration inside your infra (software to be installed on a specific machine)
- Both may (or must…) be used together to allow you to a complete dynamic and automated way to manage all your infrastructure and its configuration
- Most used tooling for Configuration Management are Ansible, Chef and Puppet
- Most of Configuration Management tools allow you to create infrastructure too but with limited capabilities

IaC: Tooling

- Two main flavors: proprietary or provider-agnostic
- Proprietary is related and managed only by a specific provider
  - Azure have ARM (Azure Resource Manager) Templates and Bicep
  - AWS has Cloud Formation
  - GCP has Google Deployment Manager
- Provider-agnostic tooling is not tightly coupled with cloud (or on-prem) provider you want to use
  - Terraform
  - Pulumi
  - Ansible

IaC: Tooling

	Pros	Cons
Proprietary	Always updated with last features Direct support from provider	Limited to one Provider You may need to learn several tools
Provider-agnostic	Better on hybrid environments Bigger Communities	Feature parity Changing Provider is not only a configuration task

Intro to Terraform

Terraform

What is Terraform

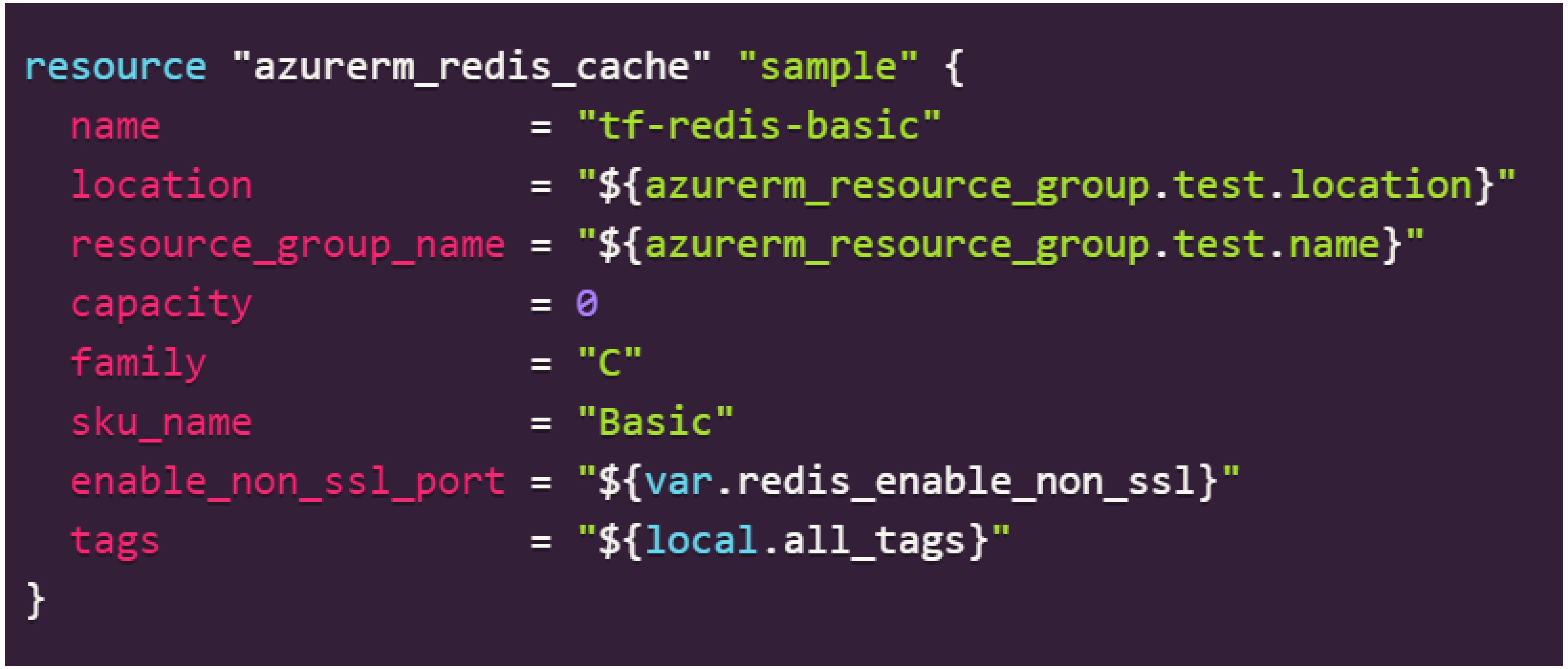
- Multi platform and multi provider IaC tooling from Hashicorp
- Biggest community with a big ecosystem of providers
- Provides a clean and easy way to write and maintain your code
- Uses a proprietary language (HCL) but similar with JSON/YAML

Terraform: How to start?

- As other DevOps tooling, you may use a CLI
- It ships as a single binary which is written in Go. Terraform is cross platform and can run on Linux, Windows, or MacOS.
- Installing terraform is easy. You simply download a zip file, unzip it, and run it.

To code? HashiCorp Configuration Language (HCL)

- The HashiCorp Configuration Language (HCL) is a small domain specific language which is based on JSON.



```
resource "azurerm_redis_cache" "sample" {
  name                  = "tf-redis-basic"
  location              = "${azurerm_resource_group.test.location}"
  resource_group_name   = "${azurerm_resource_group.test.name}"
  capacity              = 0
  family                = "C"
  sku_name              = "Basic"
  enable_non_ssl_port   = "${var.redis_enable_non_ssl}"
  tags                  = "${local.all_tags}"
}
```

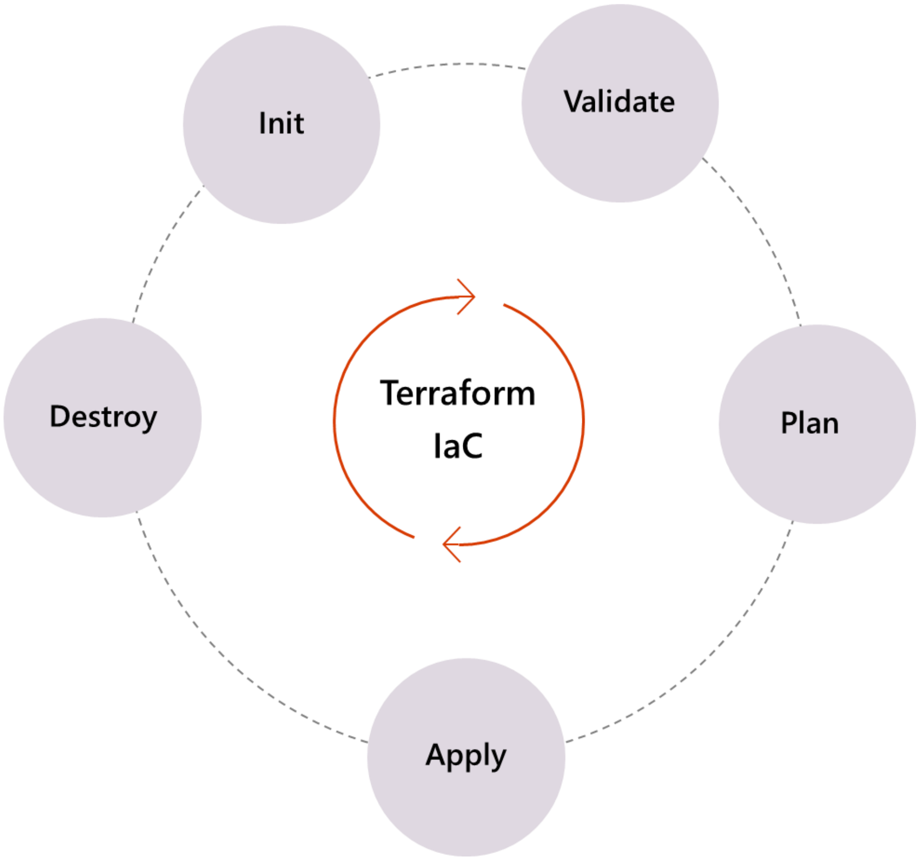
To create resources? Terraform Providers

- Big ecosystem of providers (Browse Providers | Terraform Registry)
- Allow to everyone defines your own provider if it not exists

Cloud	Infrastructure	Network	Version Control	Monitoring	Database	Etc.
Azure	Kubernetes	DNS	GitHub	DataDog	InfluxDB	
AWS	Docker	Cloudflare	GitLab	Grafana	MySQL	
Google	Rancher	HTTP	Bitbucket	PagerDuty	PostgreSQL	
AliCloud						
OpenStack						

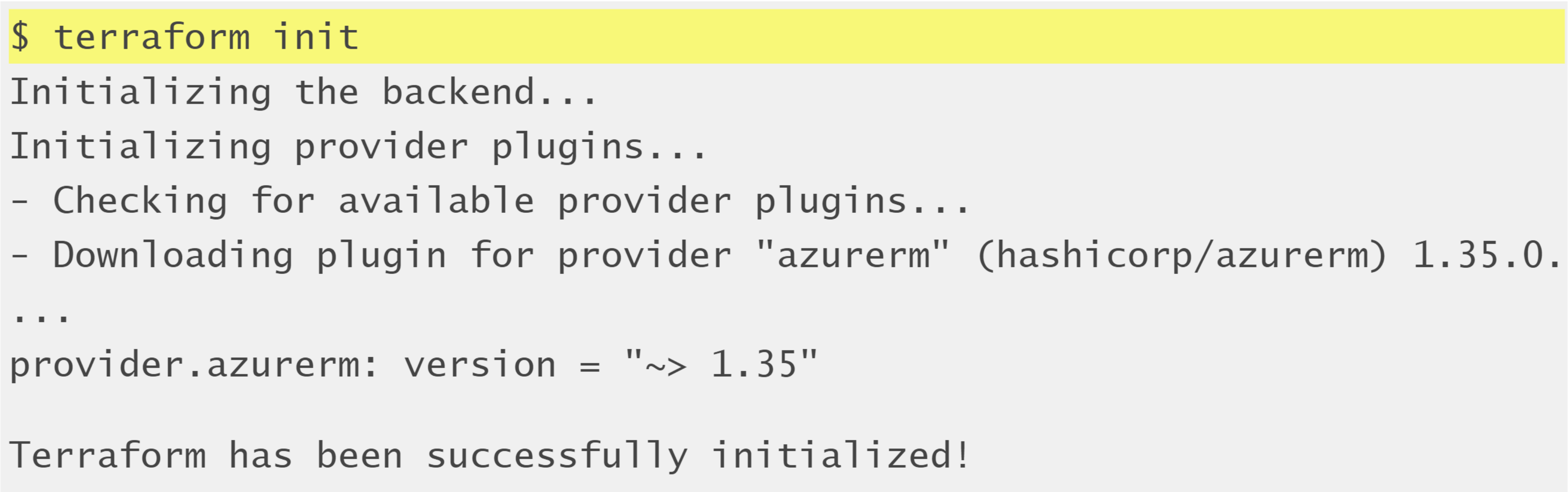
Terraform basic workflow

- Init: Initialize a working directory with Terraform configuration files
- Validate: Validates configuration files in a directory without checking remotely
- Plan: It creates an execution plan (aka WhatIf)
- Apply: Deploy the changes required to reach the desired state
- Destroy: Remove the TF manage infrastructure



Terraform Init

- Terraform fetches any required providers and modules and stores them in the **.terraform** directory.
- If you add, change or update your modules or providers you will need to run init again.

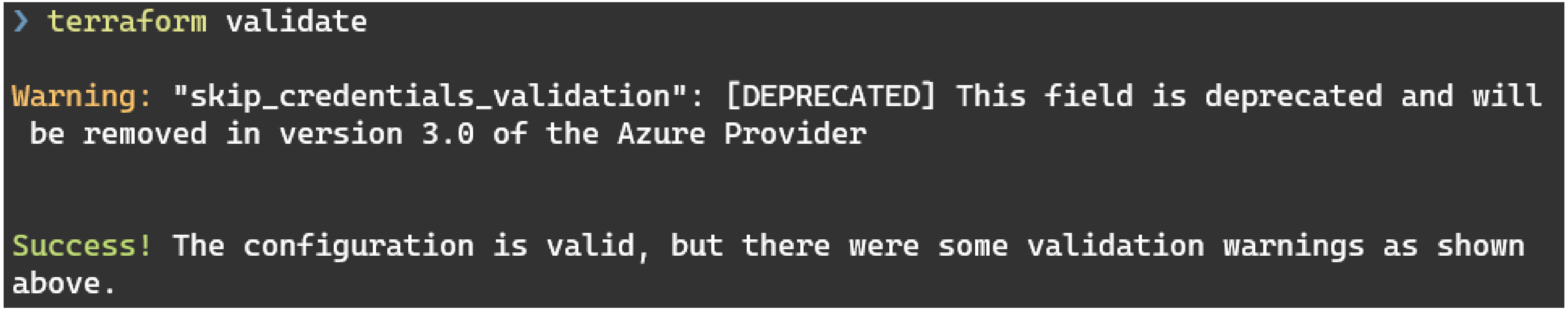


```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 1.35.0.
...
provider.azurerm: version = "~> 1.35"

Terraform has been successfully initialized!
```

Terraform Validate

- **terraform validate** checks if your terraform files are valid without making any remote validation



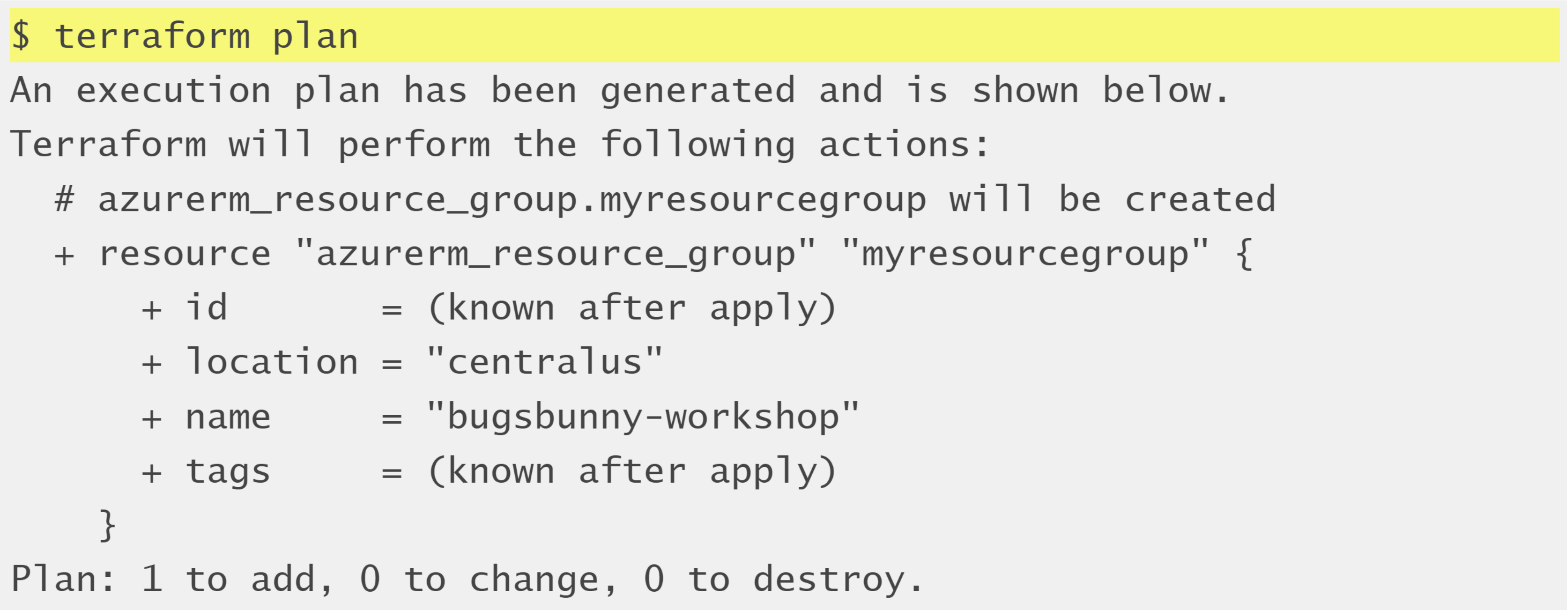
```
❯ terraform validate

Warning: "skip_credentials_validation": [DEPRECATED] This field is deprecated and will
 be removed in version 3.0 of the Azure Provider


Success! The configuration is valid, but there were some validation warnings as shown
above.
```

Terraform Plan

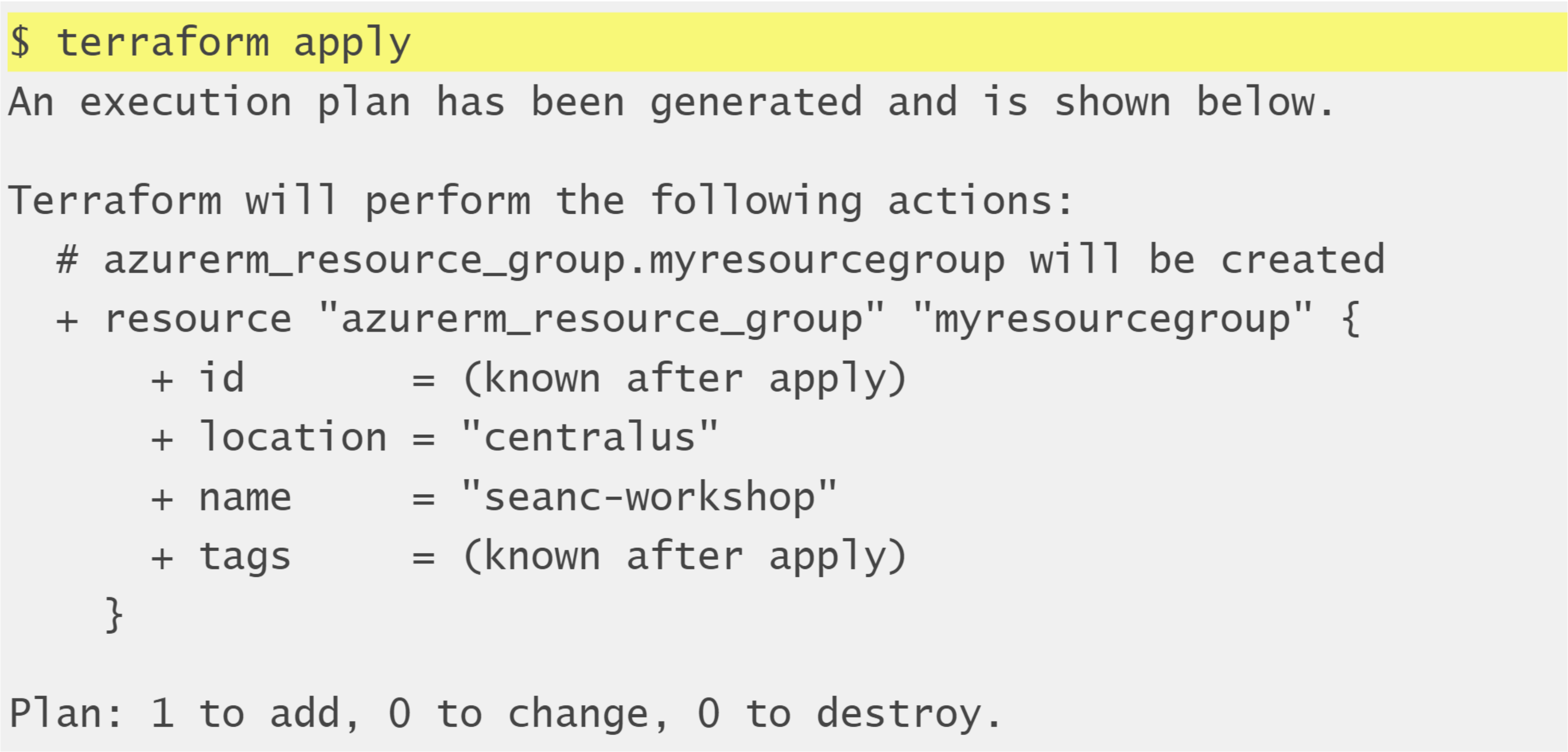
- Preview your changes with **terraform plan** before you apply them.



```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # azurerm_resource_group.myresourcegroup will be created
  + resource "azurerm_resource_group" "myresourcegroup" {
      + id       = (known after apply)
      + location = "centralus"
      + name     = "bugsbunny-workshop"
      + tags     = (known after apply)
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform Apply

- **terraform plan** runs a plan and then if you approve, it applies the changes



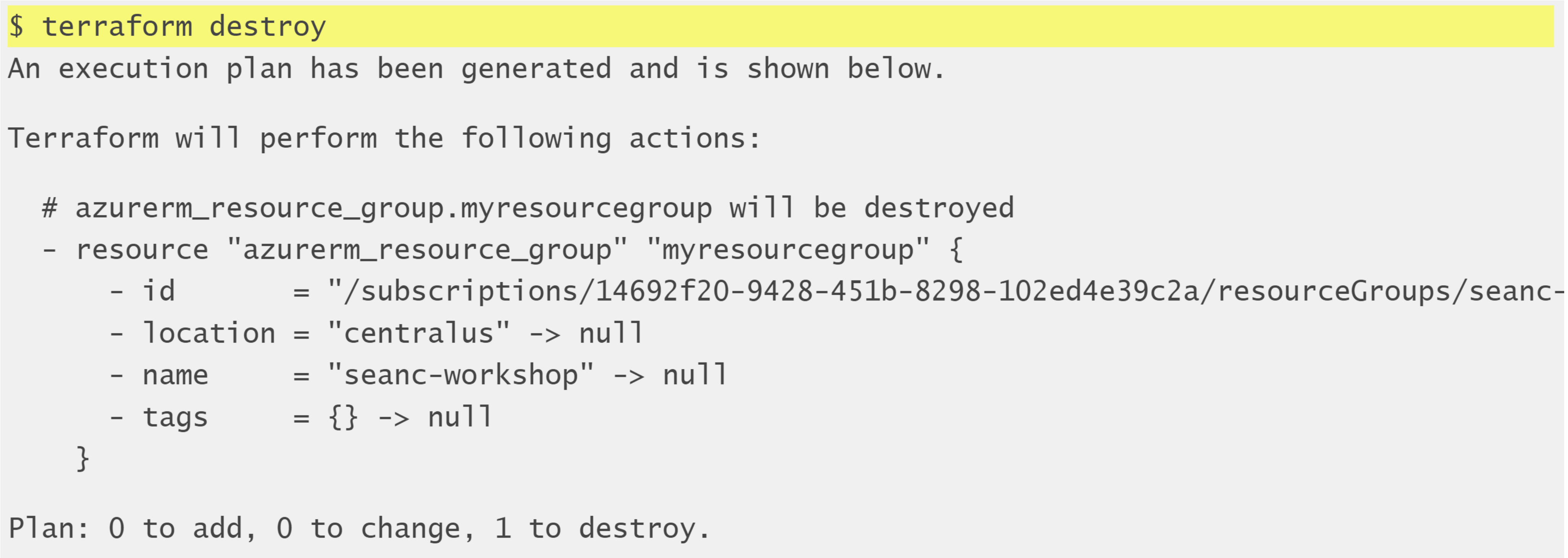
```
$ terraform apply
An execution plan has been generated and is shown below.

Terraform will perform the following actions:
  # azurerm_resource_group.myresourcegroup will be created
  + resource "azurerm_resource_group" "myresourcegroup" {
      + id       = (known after apply)
      + location = "centralus"
      + name     = "seanc-workshop"
      + tags     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform Destroy

- **terraform destroy** does the opposite. If you approve, your infrastructure is destroyed.



```
$ terraform destroy
An execution plan has been generated and is shown below.

Terraform will perform the following actions:

  # azurerm_resource_group.myresourcegroup will be destroyed
  - resource "azurerm_resource_group" "myresourcegroup" {
      - id       = "/subscriptions/14692f20-9428-451b-8298-102ed4e39c2a/resourceGroups/seanc-
      - location = "centralus" -> null
      - name     = "seanc-workshop" -> null
      - tags     = {} -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.
```

Demo – Run Terraform commands

Terraform