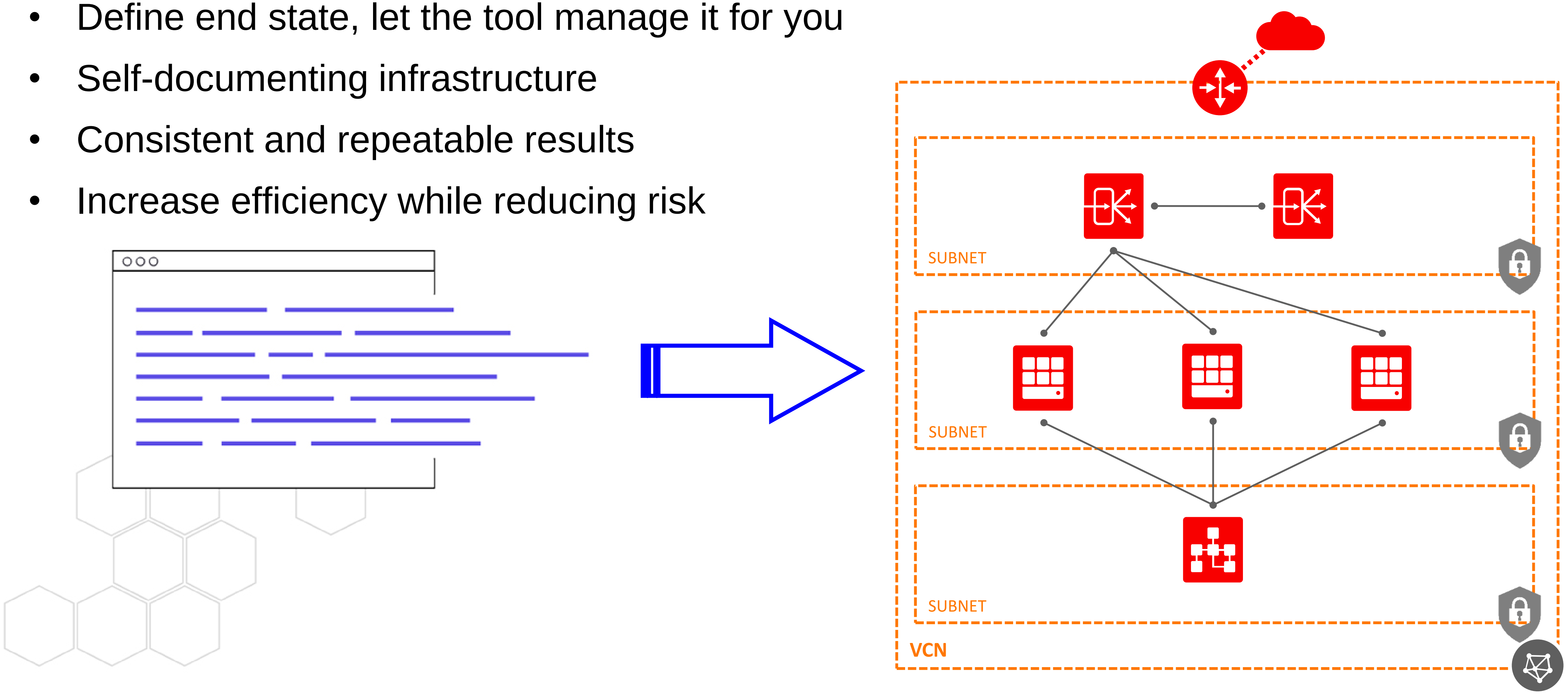
Using Terraform and Resource Manager

Agenda

- Why Infrastructure as Code
    - Best Practices
- Key Terraform Functionality
    - Configure the provider
    - Using Plan, Apply, Destroy
    - Data Sources, Resources
    - Useful Features
- Introduction to Resource Manager

Why Infrastructure as Code

- Define end state, let the tool manage it for you
- Self-documenting infrastructure
- Consistent and repeatable results
- Increase efficiency while reducing risk



IaC Best Practices – Use a good IDE

- Make it easy on yourself when writing / editing HCL
- Sublime – use Package Control to install Terraform plugin
- Atom has several linters, formatters, and auto-completion plugins available
- Visual Studio Code includes a few Terraform modules as well

IaC Best Practices - Source Control

- Treat Terraform configuration like source code
- Store in a secure location
- Limit access based on requirements
- Evaluate changes (pull request)
- Audit changes regularly

ORACLE

Terraform Configuration Files

- Configuration can be in a single file or split across multiple files.
- Terraform will merge all files with extensions **.tf** or **.tf.json** in current working directory.
- Sub-folders are not included (non-recursive).
- Files are merged in alphabetical order; resource definition sequence does not matter.
- Any files with a different extension are ignored (e.g. .jpg)

TIP: Create a top-level directory to host all your terraform files/folders/modules.

Terraform Configuration Files

The two supported configuration file formats (.tf and .tf.json) are for different audiences:

- **Humans** - The .tf format, is broadly human-readable, allows inline comments, and is generally recommended if humans are crafting your configuration.

- **Machines** - The .tf.json format is pure JSON and is meant for machine interactions, where a machine is building your configuration files.

```
keyword1 "some_name" {
  key = "value"
  nested {
     key = "value'
   }
```

```
{
  "keyword1": [
    {
      "some_name": [
        {
          "key": "value",
          "nested": [
            {
              "key": "value"
            }
          ]
        }
      ]
    }
  ]
}
```

Terraform Configuration Files – OCI provider

Providers abstract the APIs from any given third party in order to create infrastructure. Here is an OCI example leveraging an Identity and Access Management (IAM) user:

```
provider "oci" {
  tenancy_ocid     = "${var.tenancy_ocid}"
  user_ocid        = "${var.user_ocid}"
  fingerprint      = "${var.fingerprint}"
  private_key_path = "${var.private_key_path}"
  region           = "${var.region}"
}
```

The OCI provider enables Terraform to create, manage, and destroy resources within your OCI tenancy.

Terraform Configuration Files – OCI provider

The OCI provider also supports OCI Resource Principals.

If you are running Terraform on a compute instance that is a member of an IAM Dynamic Group, you can instruct Terraform to reference the resource principal:

```
provider "oci" {
  auth                 = “InstancePrincipal”
  region               = "${var.region}"
}
```

Terraform Configuration Files – Resources

Resources - Once a provider is configured we can start using that providers resources.

With the oci provider, we can start creating instances, block and object storage, networks, etc.

The following example starts an instance:

Component **Provider** **Resource Type** **Name**

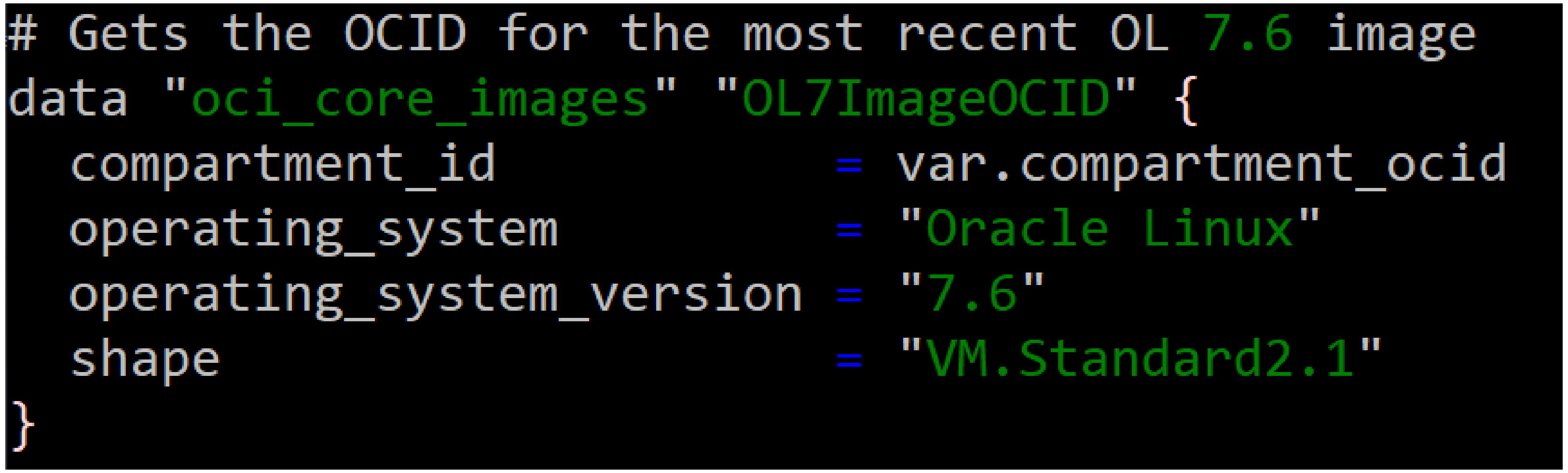
```
resource "oci_core_instance" "TFInstance" {
  availability_domain = data.oci_identity_availability_domains.ADs.availability_domains[var.AD - 1]["name"]
  compartment_id      = var.compartment_ocid
  display_name        = "TFInstance"
  source_details {
    source_type = "image"
    source_id   = data.oci_core_images.OL7ImageOCID.images[0]["id"]
  }
  shape = var.InstanceShape
```

NOTE: The combination of type and name must be unique in your configuration.

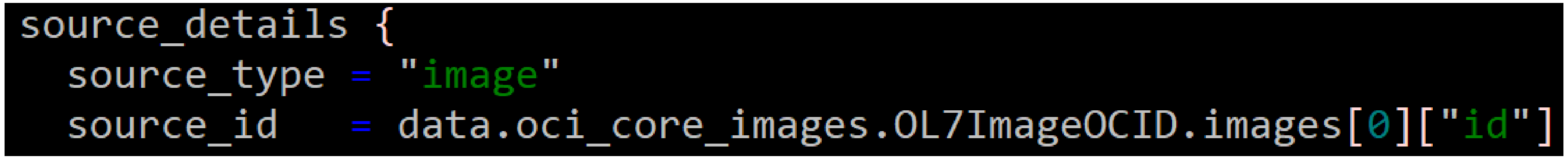
Terraform Configuration Files – Data Sources

Data Sources are used to look up information about existing resources or environment parameters (like availability domain labels or image OCIDs).

The following example returns an array of Image OCIDs that match the given input parameters.



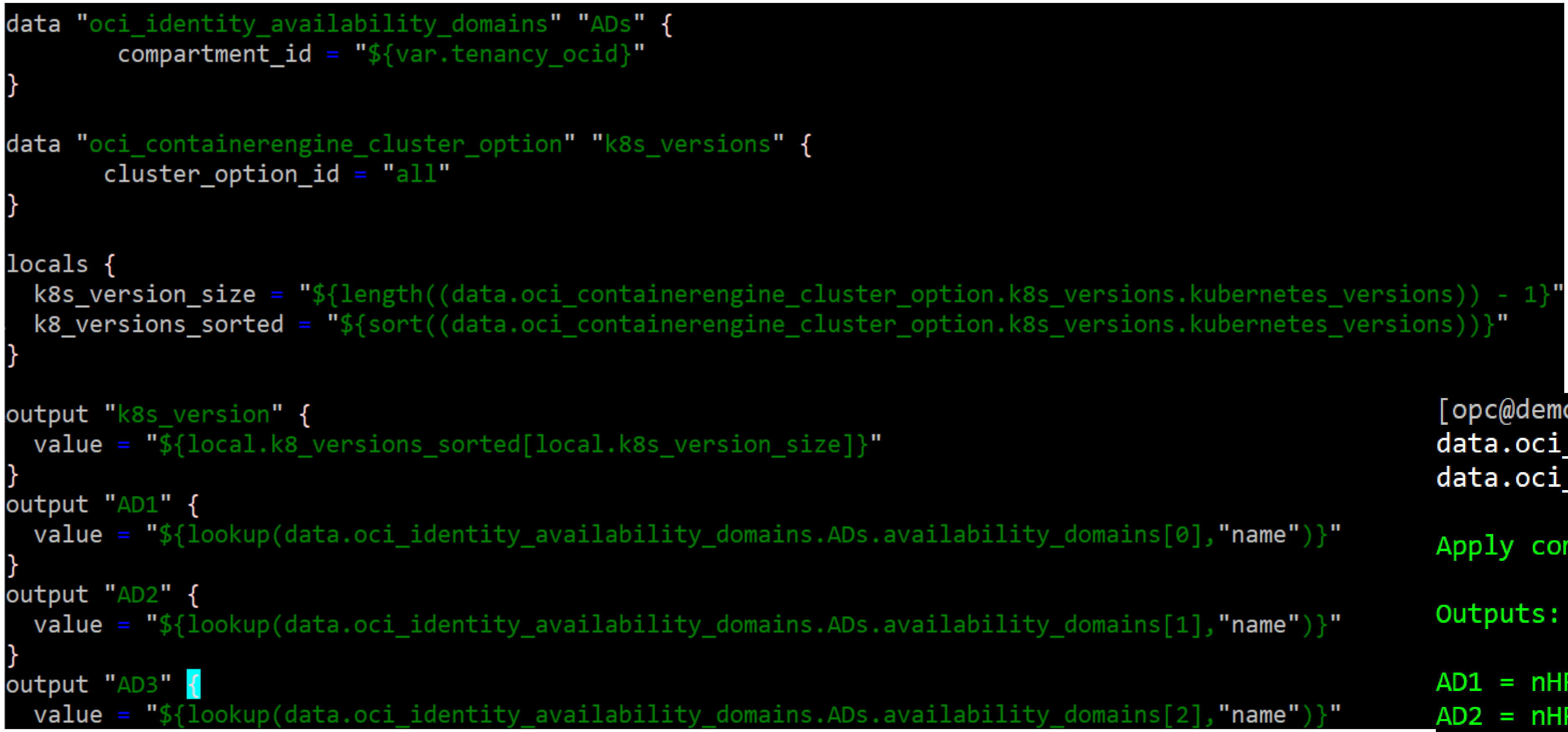
```
# Gets the OCID for the most recent OL 7.6 image
data "oci_core_images" "OL7ImageOCID" {
  compartment_id           = var.compartment_ocid
  operating_system         = "Oracle Linux"
  operating_system_version = "7.6"
  shape                    = "VM.Standard2.1"
}
```



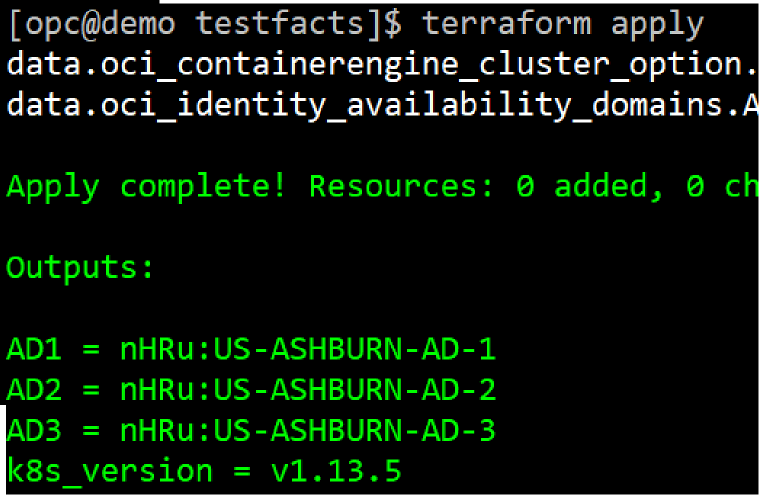
```
source_details {
  source_type = "image"
  source_id   = data.oci_core_images.OL7ImageOCID.images[0]["id"]
```

Terraform Configuration Files – Data Sources

Another example here shows how to obtain Availability Domain names (the 4-digit prefix is unique per tenancy) and the latest cluster version available for Oracle Container Engine for Kubernetes (OKE)



```
data "oci_identity_availability_domains" "ADs" {
        compartment_id = "${var.tenancy_ocid}"
}

data "oci_containerengine_cluster_option" "k8s_versions" {
       cluster_option_id = "all"
}

locals {
  k8s_version_size = "${length((data.oci_containerengine_cluster_option.k8s_versions.kubernetes_versions)) - 1}"
  k8_versions_sorted = "${sort((data.oci_containerengine_cluster_option.k8s_versions.kubernetes_versions))}"
}

output "k8s_version" {
  value = "${local.k8_versions_sorted[local.k8s_version_size]}"
}
output "AD1" {
  value = "${lookup(data.oci_identity_availability_domains.ADs.availability_domains[0],"name")}"
}
output "AD2" {
  value = "${lookup(data.oci_identity_availability_domains.ADs.availability_domains[1],"name")}"
}
output "AD3" {
  value = "${lookup(data.oci_identity_availability_domains.ADs.availability_domains[2],"name")}"
```



```
[opc@demo testfacts]$ terraform apply
data.oci_containerengine_cluster_option.
data.oci_identity_availability_domains.A

Apply complete! Resources: 0 added, 0 ch

Outputs:

AD1 = nHRu:US-ASHBURN-AD-1
AD2 = nHRu:US-ASHBURN-AD-2
AD3 = nHRu:US-ASHBURN-AD-3
k8s_version = v1.13.5
```

Terraform Configuration Files – Building your first resource

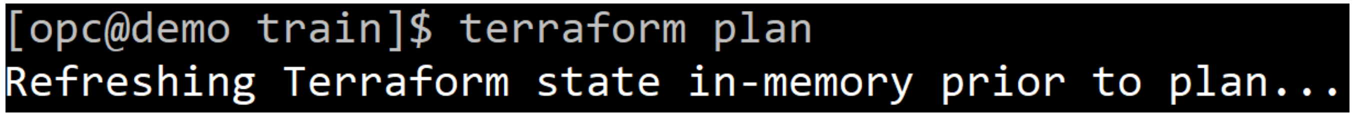
Lets start by defining a Virtual Cloud Network (VCN). For now we will hard-code all values. In a later session we will discuss variables.

Example:

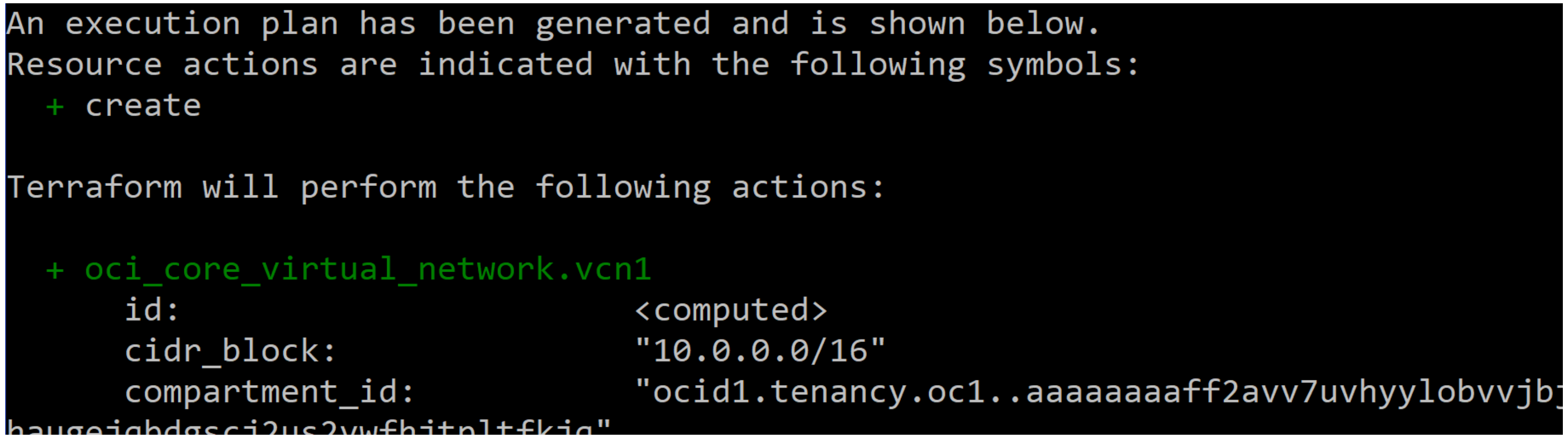
```
resource "oci_core_virtual_network" “ExampleVCN" {
  cidr_block = "10.0.0.0/16"
  dns_label = “TFExampleVCN"
  compartment_id = “ocid1.compartment.oc1..aaaaaaaa72374732”
  display_name = “tfexamplevcn”
}
```

Terraform Actions – Plan

- **terraform plan** shows you what will happen when you execute your configuration
- Plan output can be saved and used during execution if you are testing several potential scenarios
- The plan will show reasons for certain actions (such as re-create if compute image OCID is changed)



```
[opc@demo train]$ terraform plan
Refreshing Terraform state in-memory prior to plan...
```



```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + oci_core_virtual_network.vcn1
      id:                          <computed>
      cidr_block:                  "10.0.0.0/16"
      compartment_id:              "ocid1.tenancy.oc1..aaaaaaaaff2avv7uvhyylobvvjbj
haugeiqhdgsci2us2vwfhitnltfkiq"
```

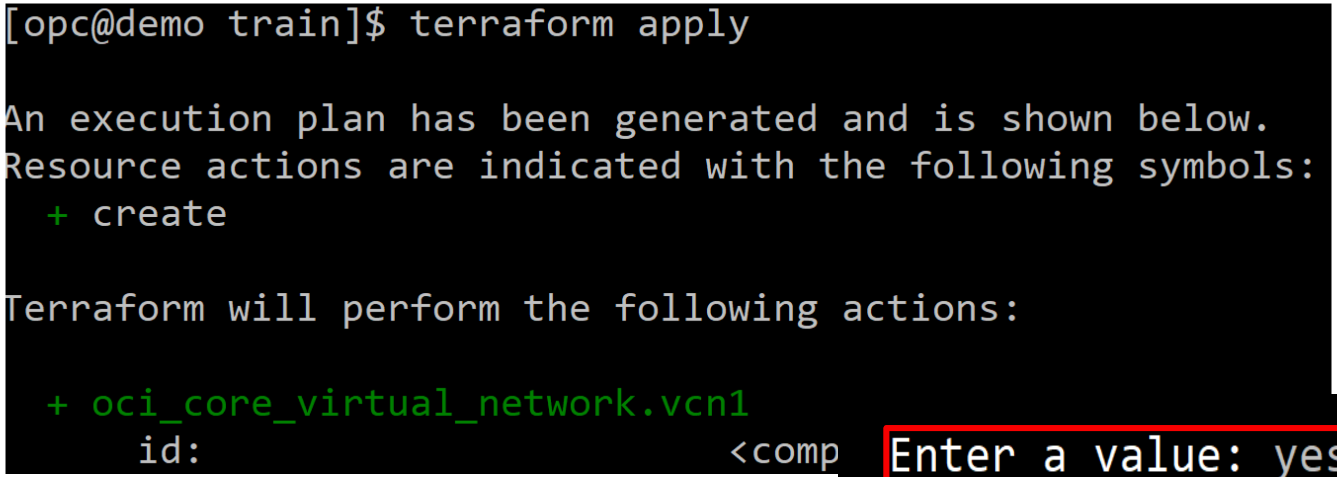
Terraform Actions – Plan: Indicators

+	indicates a resource will be created
-	indicates a resource will be destroyed
~	indicates a resource will be updated in-place
-/+	indicates a resources will be destroyed and re-created

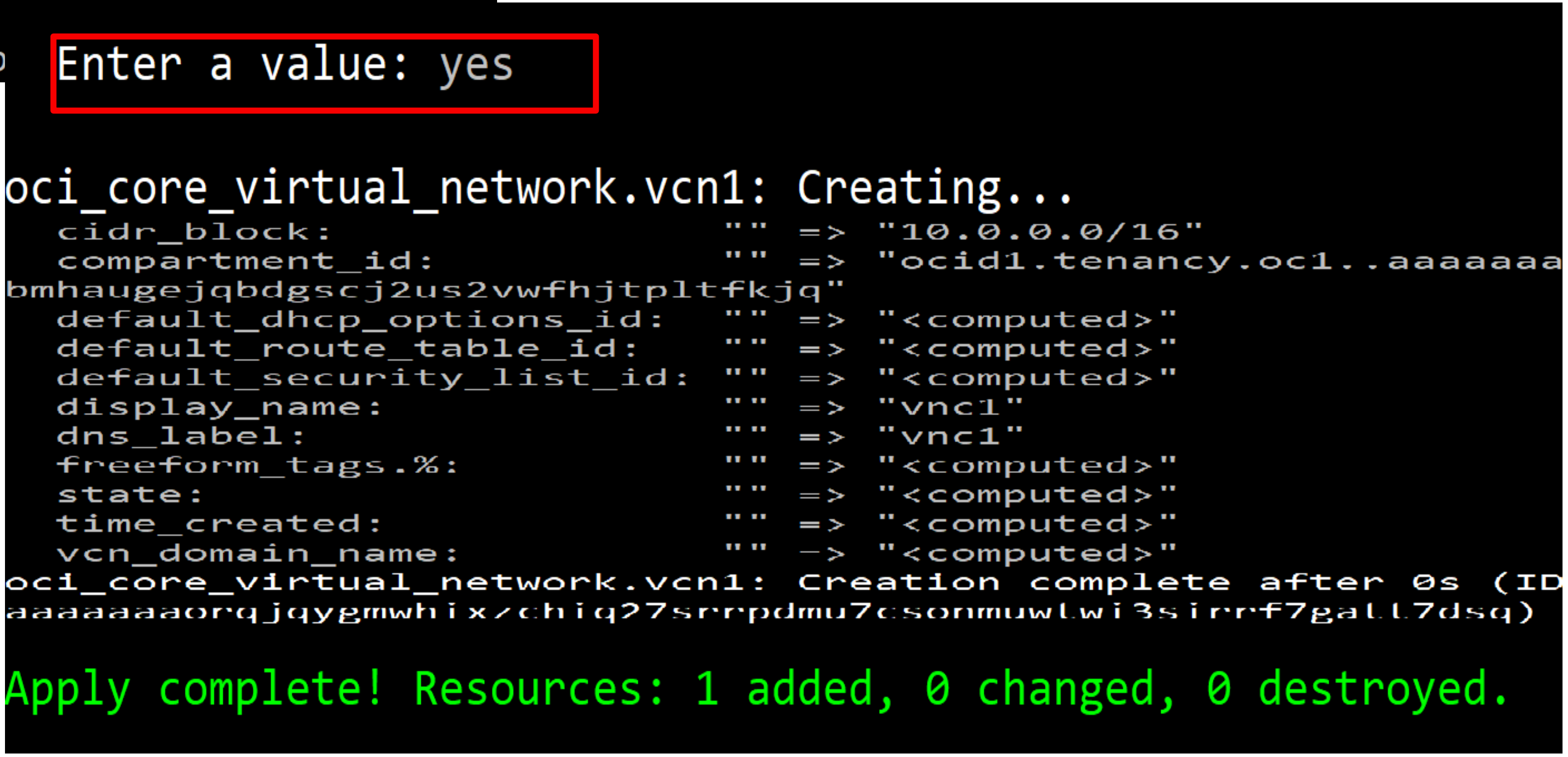
Terraform Actions – Apply

- **terraform apply** will execute the definitions in your configuration file
- Creates / Modifies / Deletes resources in order based on dependencies
- Parallelizes changes where possible
- Updates existing resources when updates are allowed
- Deletes and re-creates existing resources when an update is not allowed
    - such as a change to the image OCID defined for an instance

Terraform Actions – Apply: Example



```
[opc@demo train]$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + oci_core_virtual_network.vcn1
      id:                          <comp
```



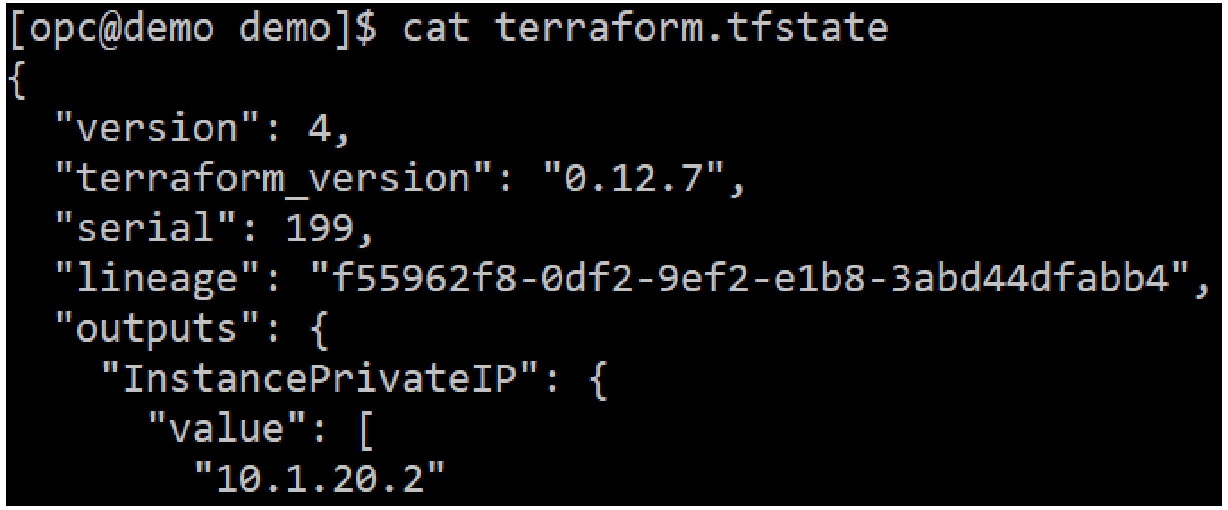
```
  Enter a value: yes

oci_core_virtual_network.vcn1: Creating...
  cidr_block:                 "" => "10.0.0.0/16"
  compartment_id:             "" => "ocid1.tenancy.oc1..aaaaaaa
bmhaugejqbdgscj2us2vwfhjtpltfkjq"
  default_dhcp_options_id:    "" => "<computed>"
  default_route_table_id:     "" => "<computed>"
  default_security_list_id:   "" => "<computed>"
  display_name:               "" => "vnc1"
  dns_label:                  "" => "vnc1"
  freeform_tags.%:            "" => "<computed>"
  state:                      "" => "<computed>"
  time_created:               "" => "<computed>"
  vcn_domain_name:            "" -> "<computed>"
oci_core_virtual_network.vcn1: Creation complete after 0s (ID
aaaaaaaorqjqygmwhix/chiq27srrpdmu7csonmuwlwi3sirrf7gall7dsq)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Managing the State of the Environment

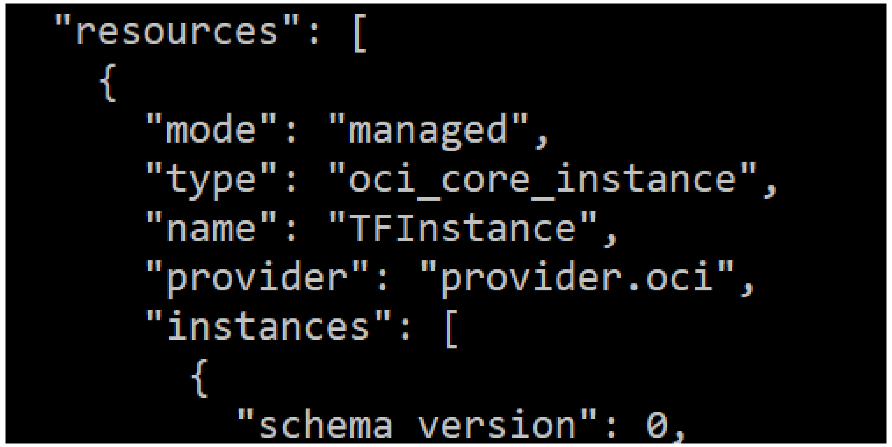
- Terraform stores the state of your managed infrastructure from the last time **terraform apply** was executed.
- Terraform uses this state to create plans and evaluate potential changes to your infrastructure.
- It is critical that this state is maintained appropriately so future executions operate as expected.

Terraform Local State

- State is stored locally on local machine in JSON format
- Because it must exist, it is a frequent source of merge conflicts
- It is generally acceptable for individuals and small teams
- Tends not to scale for large teams (see OCI Resource Manager)
- Requires a more "mono repo" pattern



```
[opc@demo demo]$ cat terraform.tfstate
{
  "version": 4,
  "terraform_version": "0.12.7",
  "serial": 199,
  "lineage": "f55962f8-0df2-9ef2-e1b8-3abd44dfabb4",
  "outputs": {
    "InstancePrivateIP": {
      "value": [
        "10.1.20.2"
```

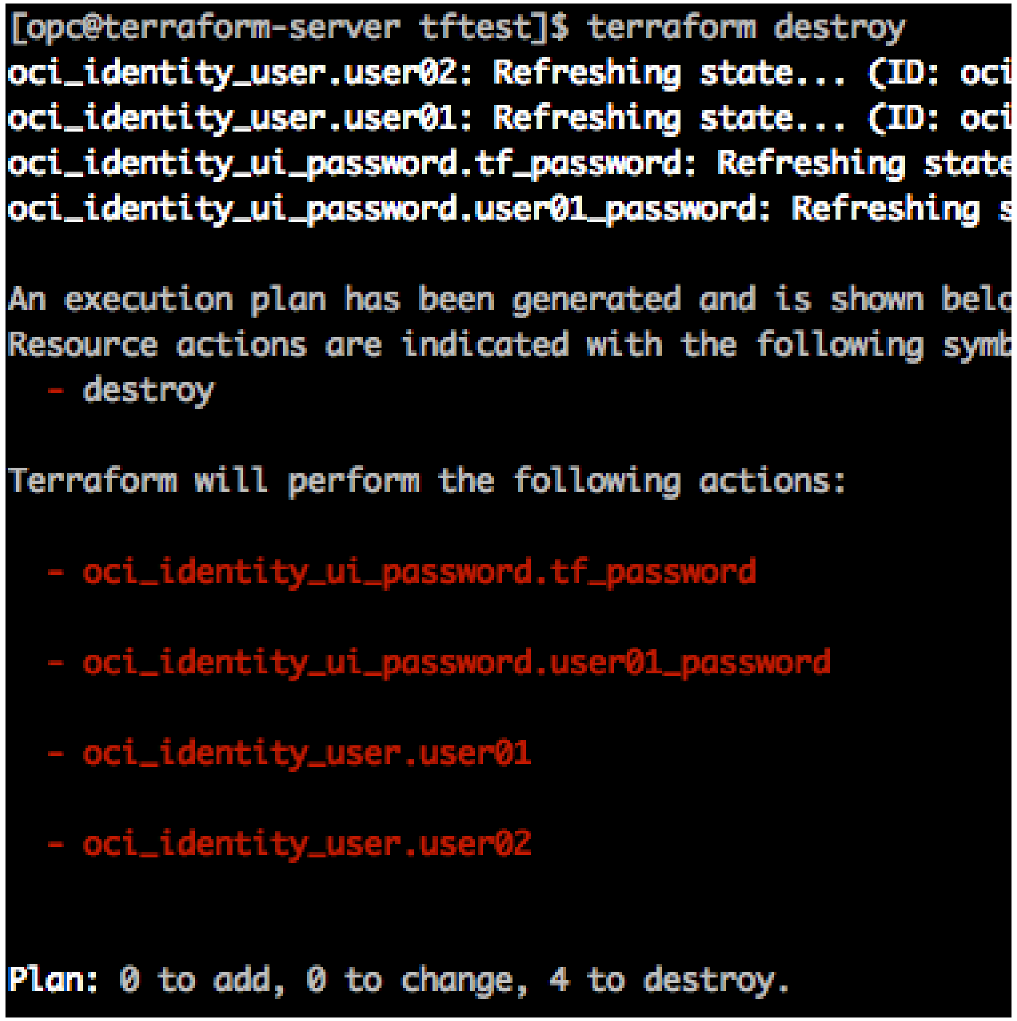


```
  "resources": [
    {
      "mode": "managed",
      "type": "oci_core_instance",
      "name": "TFInstance",
      "provider": "provider.oci",
      "instances": [
        {
          "schema_version": 0,
```

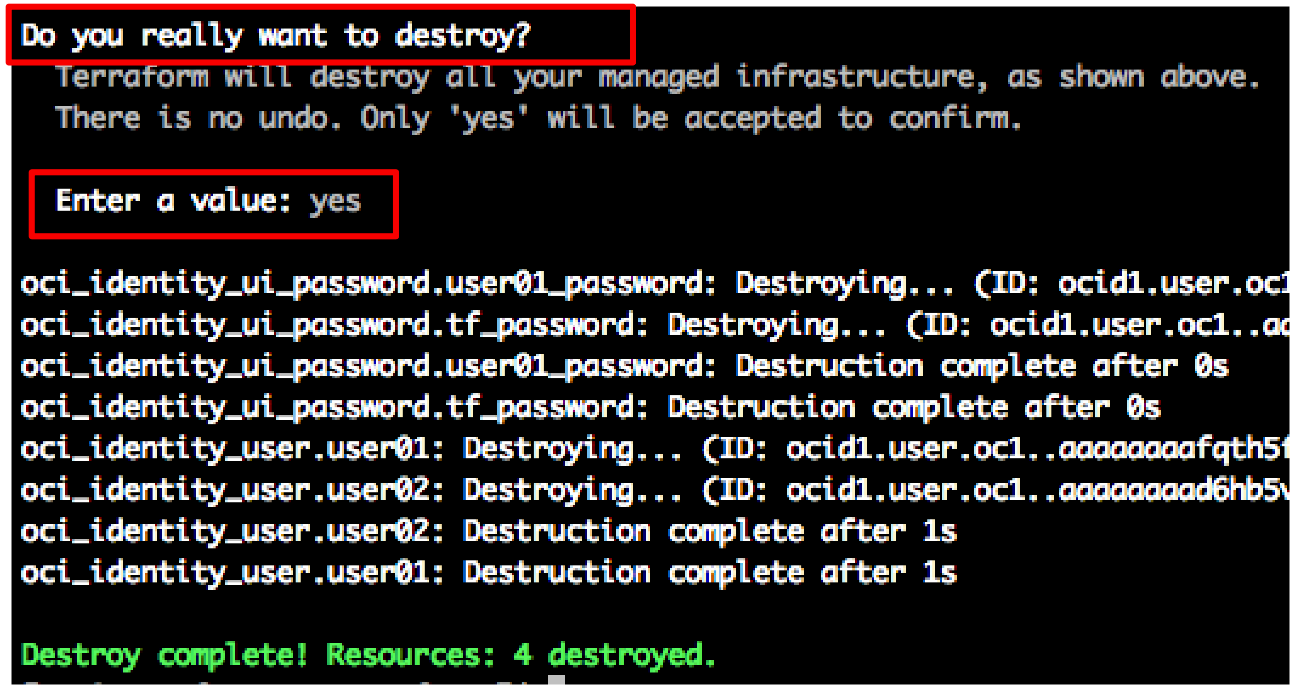
Terraform Actions – Destroy

- Destroys running infrastructure known to the current state file
- Does not touch infrastructure not managed by Terraform
- Terraform destroy will ask for permission, requiring an explicit "yes" as input.
- The terraform destroy command, without any options, destroys everything!
- If you only want to destroy a specific resource then you can use the **`-target`** flag.

Terraform Actions – Destroy: Example



```
[opc@terraform-server tftest]$ terraform destroy
oci_identity_user.user02: Refreshing state... (ID: oci
oci_identity_user.user01: Refreshing state... (ID: oci
oci_identity_ui_password.tf_password: Refreshing state
oci_identity_ui_password.user01_password: Refreshing s

An execution plan has been generated and is shown belo
Resource actions are indicated with the following symb
  - destroy

Terraform will perform the following actions:

  - oci_identity_ui_password.tf_password

  - oci_identity_ui_password.user01_password

  - oci_identity_user.user01

  - oci_identity_user.user02


Plan: 0 to add, 0 to change, 4 to destroy.
```



```
Do you really want to destroy?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

oci_identity_ui_password.user01_password: Destroying... (ID: ocid1.user.oc1
oci_identity_ui_password.tf_password: Destroying... (ID: ocid1.user.oc1..aa
oci_identity_ui_password.user01_password: Destruction complete after 0s
oci_identity_ui_password.tf_password: Destruction complete after 0s
oci_identity_user.user01: Destroying... (ID: ocid1.user.oc1..aaaaaaaafqth5f
oci_identity_user.user02: Destroying... (ID: ocid1.user.oc1..aaaaaaaad6hb5v
oci_identity_user.user02: Destruction complete after 1s
oci_identity_user.user01: Destruction complete after 1s

Destroy complete! Resources: 4 destroyed.
```

Terraform Actions – Roll Back/Forward

- Terraform does not roll back or forward when execution fails
- When failure on **apply,** terraform will simply exit. All resources created up to that point are left to exist. The state file will be updated accordingly.
- When failure on **destroy**, terraform will stop deleting resources. All resources deleted up to that point are gone and cannot be recovered.

- When creation of a new resource stack fails, usually best to roll forward. Correct the error in the configuration and apply again.
  - Terraform is idempotent; resources already created (tracked in the state file) will not be affected.

Terraform variables

- Like all good source code, you should try to avoid hard-coding values
- Terraform supports environment variables, defined variables (file), and run-time variables

Variables can be **string**, **list**, **boolean** and **map**.

String Variables

```
# Choose an Availability Domain
variable "AD" {
  default = "1"
}
variable "InstanceShape" {
  default = "VM.Standard2.2"
}

variable "InstanceImageDisplayName" {
    default = "Oracle-Linux-7.6-2019.7.18-0"
}
```

Map Variable

```
variable "environment" { default = "dev" }
variable "shape" {
   type = "map"
   default = {
     dev = "VM.Standard2.2"
     test = "VM.Standard2.4"
     prod = "BM.Standard2.52"
   }
}

resource "oci_core_instance" "app-server" {
  image = "${var.image}"
  shape = "${lookup(var.instance_type, var.environment)}"
  subnet_id = "${var.subnet_id}"
}
```

Assigning and Overriding Variables

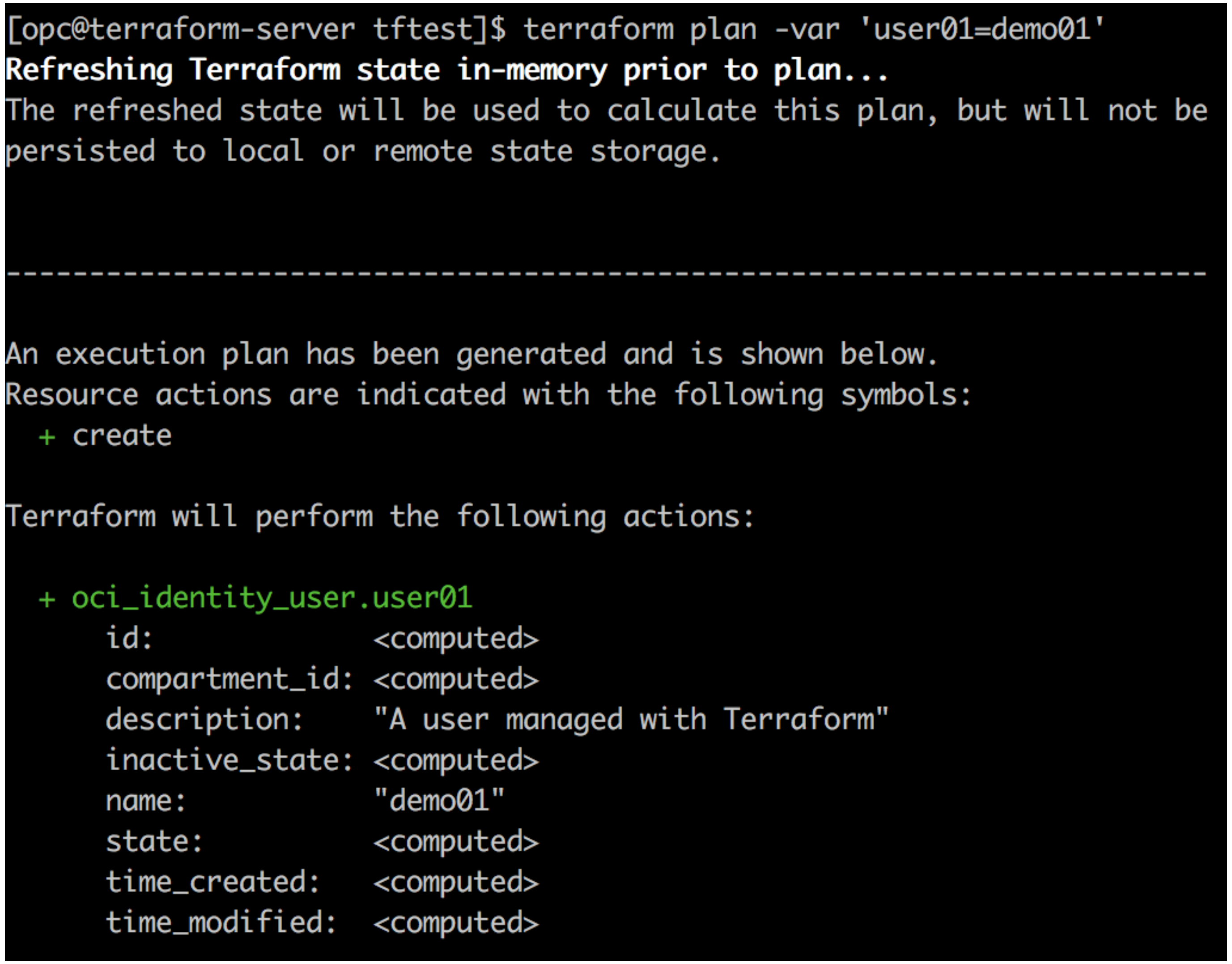
- Variables without default values have to have a value assigned.
- Variables that have no default value set will cause terraform to prompt for the variable during a plan or apply.
- Default variables can be overridden by the environment, command line, tfvars file, or inline.

  ```
  export TF_VAR_user_password=“P@ssw0rd1!”
  ```
- An example of overriding a variable from the command line is as follows:

  ```
  $ terraform apply -var 'InstanceShape=VM.Standard2.4'
  ```
- A .tfvars file can also be used to set variables and their values

  ```
  instance_type="VM.Standard2.2"
  ```

Assigning and Overriding Variables - Example



```
[opc@terraform-server tftest]$ terraform plan -var 'user01=demo01'
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.



------------------------------------------------------------------------


An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + oci_identity_user.user01
      id:                <computed>
      compartment_id:    <computed>
      description:       "A user managed with Terraform"
      inactive_state:    <computed>
      name:              "demo01"
      state:             <computed>
      time_created:      <computed>
      time_modified:     <computed>
```

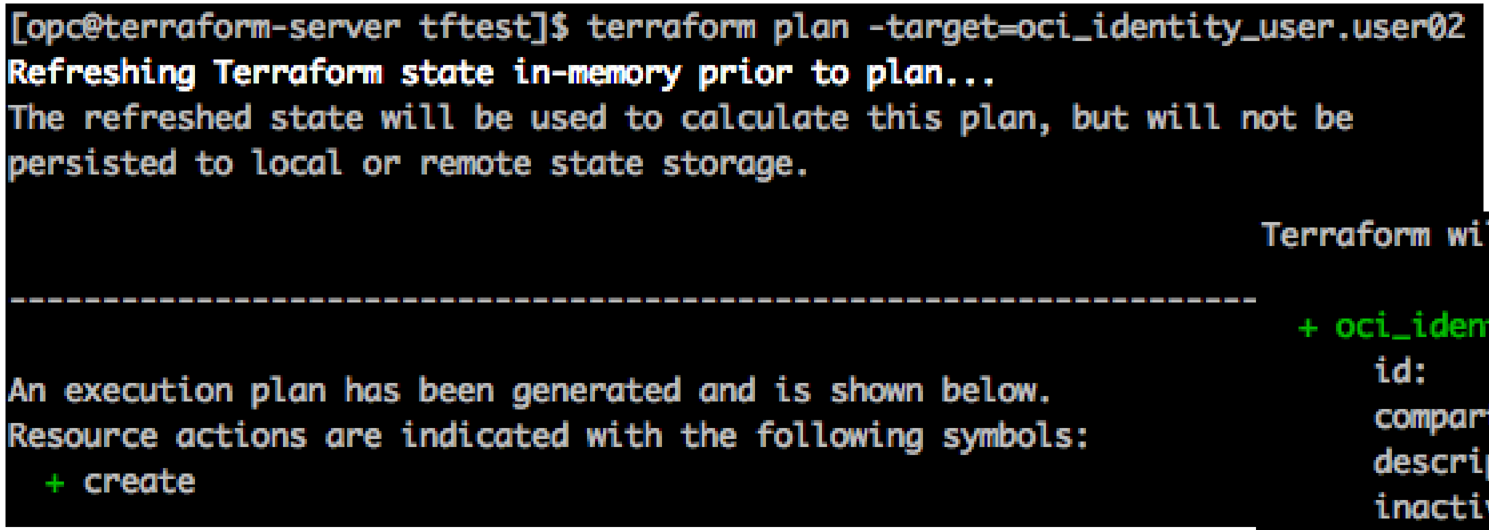
Useful Features for Managing IaC with Terraform

- Target resources during execution
- Working with outputs
- Organize multiple modules into a single stack
- Import straggler resources

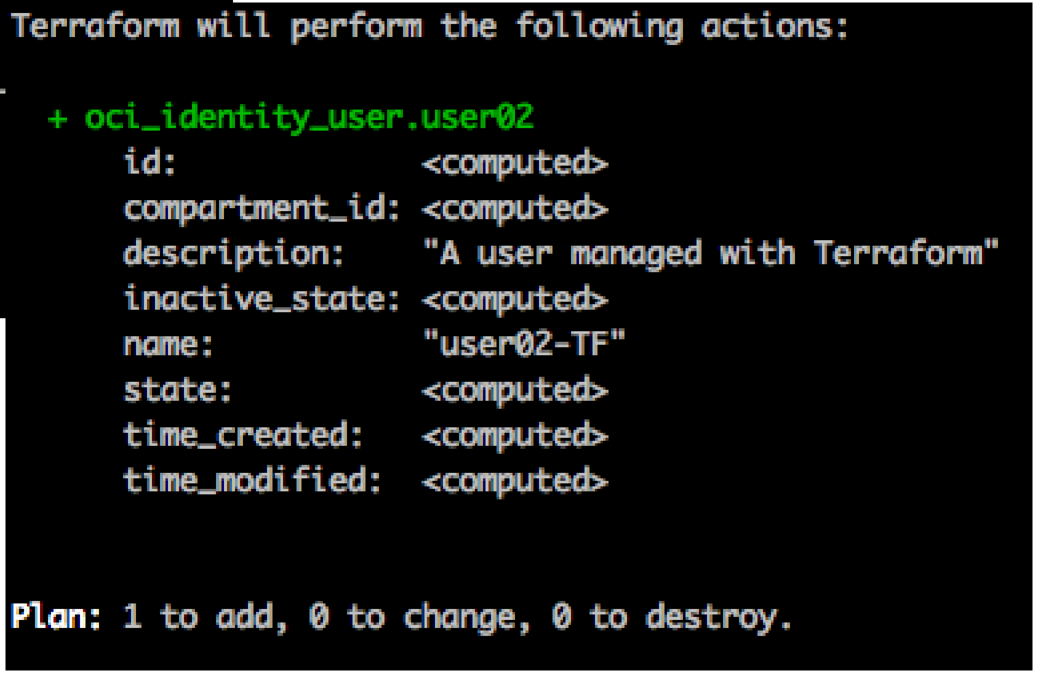
ORACLE

Terraform – Targeting resources

- You can use the `-target` flag on both the `terraform plan` and `terraform apply` commands.
- It allows you to target a resource or more if you specify multiple `-target` flags



```
[opc@terraform-server tftest]$ terraform plan -target=oci_identity_user.user02
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.


------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
```



```
Terraform will perform the following actions:

  + oci_identity_user.user02
      id:               <computed>
      compartment_id:   <computed>
      description:      "A user managed with Terraform"
      inactive_state:   <computed>
      name:             "user02-TF"
      state:            <computed>
      time_created:     <computed>
      time_modified:    <computed>


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

Terraform - Outputs

Terraform can be directed to display the variables that are generated dynamically as pat of the process of creating the infrastructure.

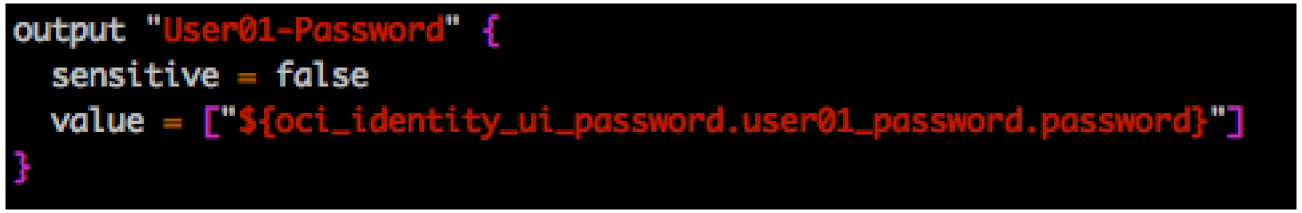
For example, after a run we might want to see the public ip of the host:

```
$ cat outputs.tf
output "InstancePrivateIP" {  value = ["${data.oci_core_vnic.InstanceVnic.private_ip_address}"]}
output "InstancePublicIP" { value = ["${data.oci_core_vnic.InstanceVnic.public_ip_address}"]}
```

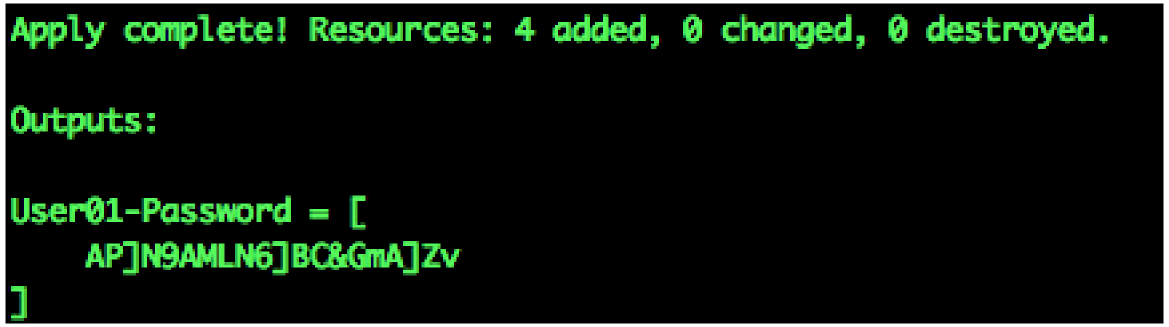
After a terraform apply:

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
State path:
Outputs:
  InstancePrivateIP = [ 10.0.0.10 ]
  InstancePublicIP = [ 129.146.3.173]
```

Terraform output - Example



```
output "User01-Password" {
  sensitive = false
  value = ["${oci_identity_ui_password.user01_password.password}"]
}
```



```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

User01-Password = [
    AP]N9AMLN6]BC&GmA]Zv
]
```

Terraform Modules

- Portable Terraform configurations (packages)
- Allow separation of concerns and responsibilities among teams
- Modules are just Terraform configurations inside a folder - there's nothing special about them.

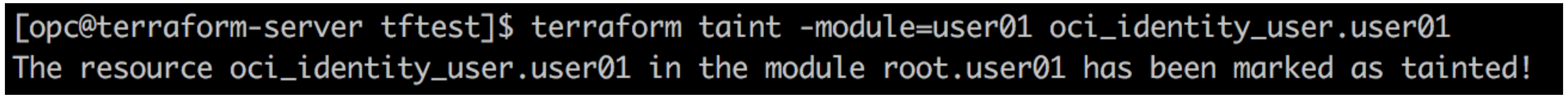
```
module "users" {
        source = "modules/users“
        compartment_ocid = var.compartment_ocid}
        tenancy_ocid = var.tenancy_ocid
        User_group = var.user_group_assign
}
```

Terraform Taint

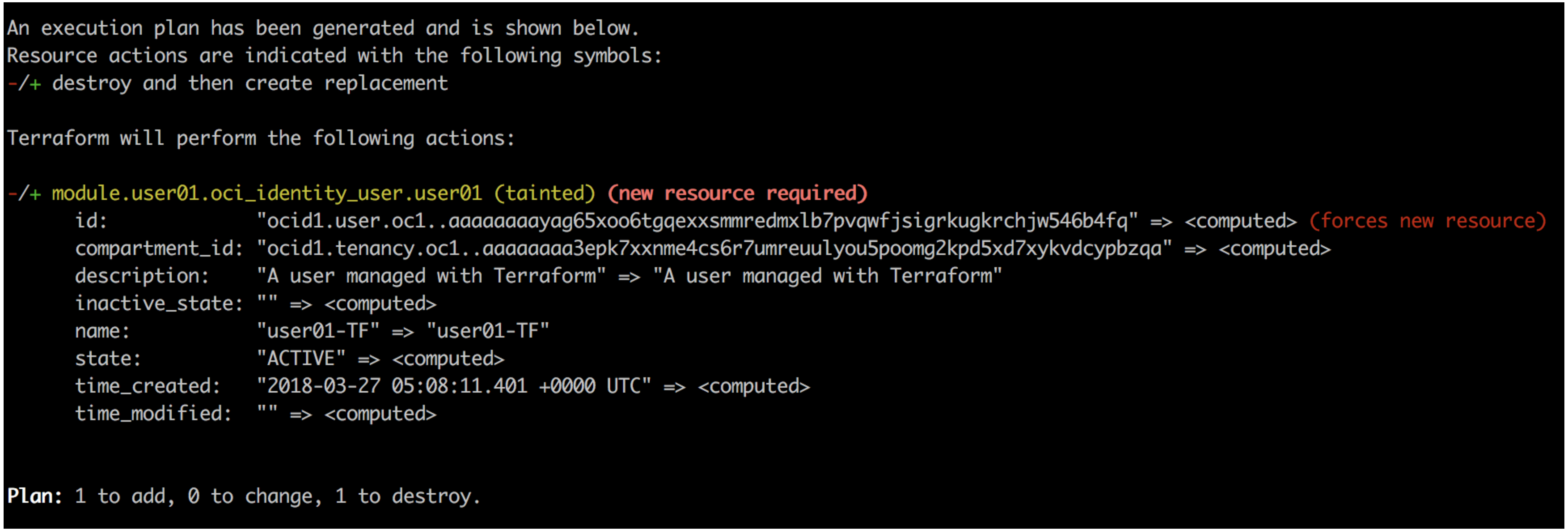
- The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.
- Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource.

```
terraform taint -module=user01 oci_identity_user.user01
```

Terraform Taint – Example



```
[opc@terraform-server tftest]$ terraform taint -module=user01 oci_identity_user.user01
The resource oci_identity_user.user01 in the module root.user01 has been marked as tainted!
```



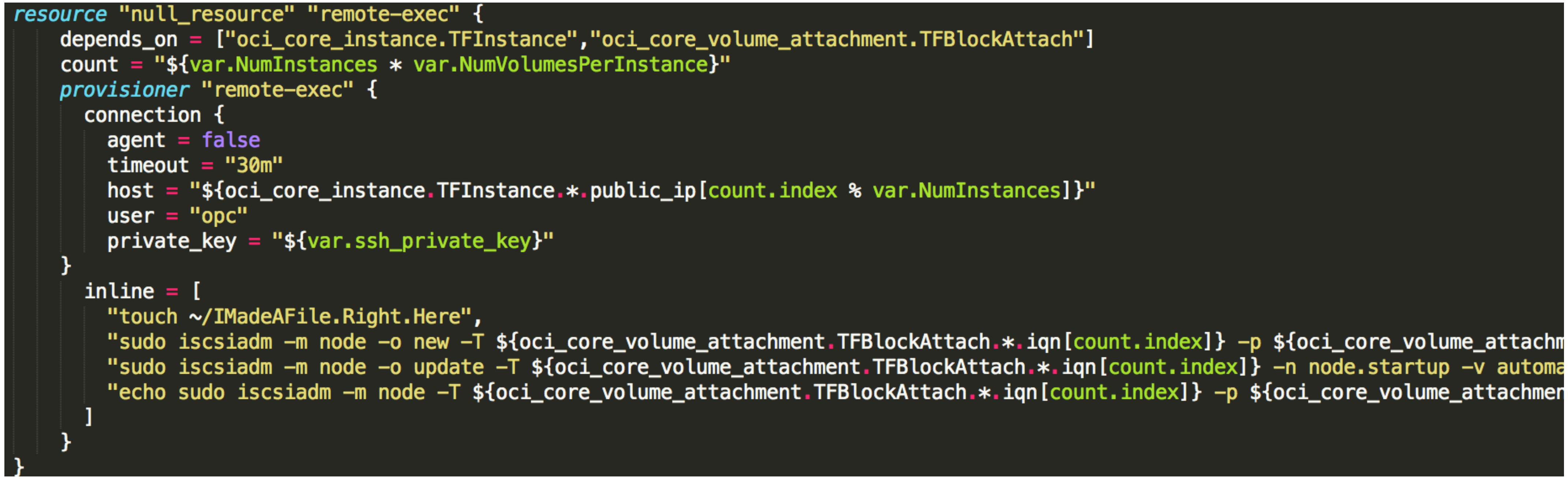
```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

-/+ module.user01.oci_identity_user.user01 (tainted) (new resource required)
      id:                "ocid1.user.oc1..aaaaaaaayag65xoo6tgqexxsmmredmxlb7pvqwfjsigrkugkrchjw546b4fq" => <computed> (forces new resource)
      compartment_id: "ocid1.tenancy.oc1..aaaaaaaa3epk7xxnme4cs6r7umreuulyou5poomg2kpd5xd7xykvdcypbzqa" => <computed>
      description:     "A user managed with Terraform" => "A user managed with Terraform"
      inactive_state: "" => <computed>
      name:            "user01-TF" => "user01-TF"
      state:           "ACTIVE" => <computed>
      time_created:    "2018-03-27 05:08:11.401 +0000 UTC" => <computed>
      time_modified:   "" => <computed>


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

Terraform Provisioners

Terraform can also integrate with provisioners like Chef, puppet, Ansible, shells scripts. An example below is using a provisioner to remote-exec a command to touch a file.

```
resource "null_resource" "remote-exec" {
  depends_on = ["oci_core_instance.TFInstance"]

  provisioner "remote-exec" {
    connection {
      agent       = false
      timeout     = "10m"
      host        = "${data.oci_core_vnic.InstanceVnic.public_ip_address}"
      user        = "opc"
      private_key = "${var.ssh_private_key}"
    }
    inline = [
      "touch ~/IMadeAFile.Right.Here",
    ]
  }
}
```

Remote-exec Provisioner – Example



```
resource "null_resource" "remote-exec" {
    depends_on = ["oci_core_instance.TFInstance","oci_core_volume_attachment.TFBlockAttach"]
    count = "${var.NumInstances * var.NumVolumesPerInstance}"
    provisioner "remote-exec" {
      connection {
        agent = false
        timeout = "30m"
        host = "${oci_core_instance.TFInstance.*.public_ip[count.index % var.NumInstances]}"
        user = "opc"
        private_key = "${var.ssh_private_key}"
    }
      inline = [
        "touch ~/IMadeAFile.Right.Here",
        "sudo iscsiadm -m node -o new -T ${oci_core_volume_attachment.TFBlockAttach.*.iqn[count.index]} -p ${oci_core_volume_attachm
        "sudo iscsiadm -m node -o update -T ${oci_core_volume_attachment.TFBlockAttach.*.iqn[count.index]} -n node.startup -v automa
        "echo sudo iscsiadm -m node -T ${oci_core_volume_attachment.TFBlockAttach.*.iqn[count.index]} -p ${oci_core_volume_attachmen
      ]
    }
}
```

Challenge – Managing State in complex environments

Managing state in an environment owned by distributed teams or individuals can be challenging. If two users try to update a set of resources controlled by Terraform, it could create problems.

So how do we solve this?

Option 1: Remote Backend for state file management

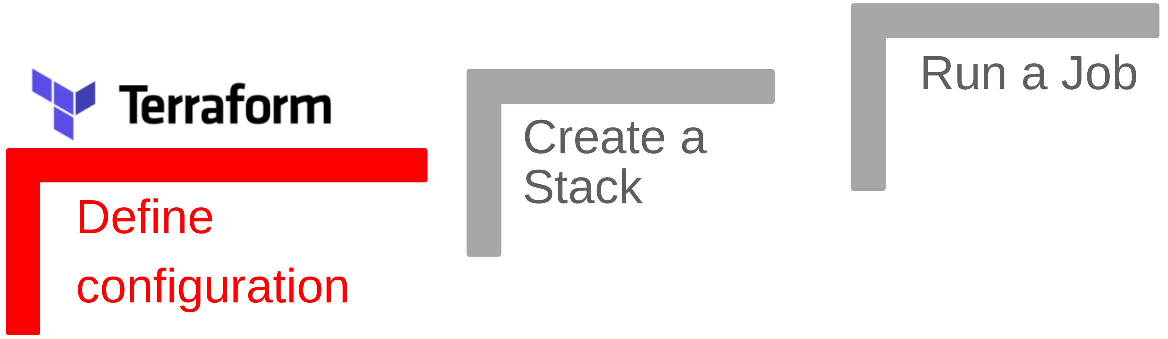
- Built-in feature; writes the state data to a remote data store (Object Storage Bucket)
- Sample configuration defining remote backend with Object Storage

Usage
```
terraform {
  backend “http” {
    address = https://objectstorage.us-phoenix-1.oraclecloud.com/p/T-Masdf/b...
  }
}
```

Reference
```
data “terraform_remote_state” “example” {
  backend = “http”
  config = {
    address = https://objectstorage.us-phoenix-1.oraclecloud.com/p/T-Masdf/b...
  }
}
```

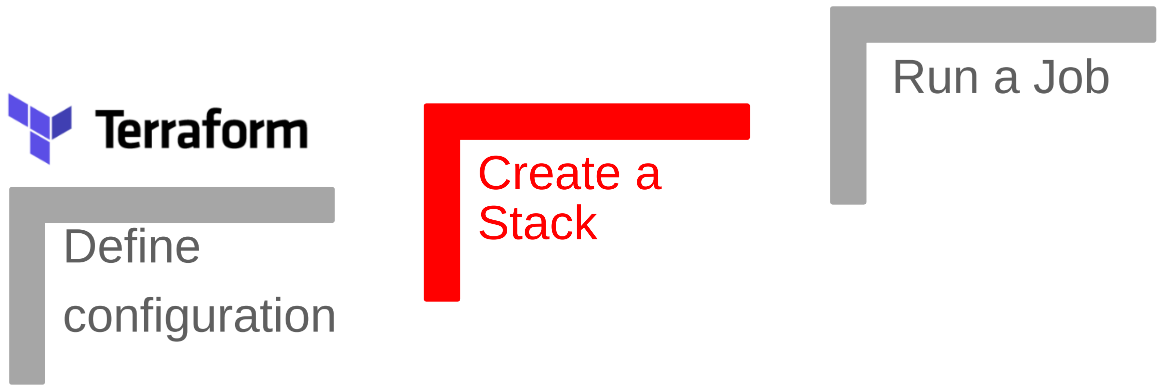
Option 2: Oracle Resource Manager

- Managed Service (Terraform as a Service)
- Resource collections defined as Stacks.
- Only one action at a time per stack; Terraform state file stored / managed by OCI
- Utilizes zip file collection of all Terrafor configuration
- Permissions controlled by IAM; no user credentials assigned to Provider

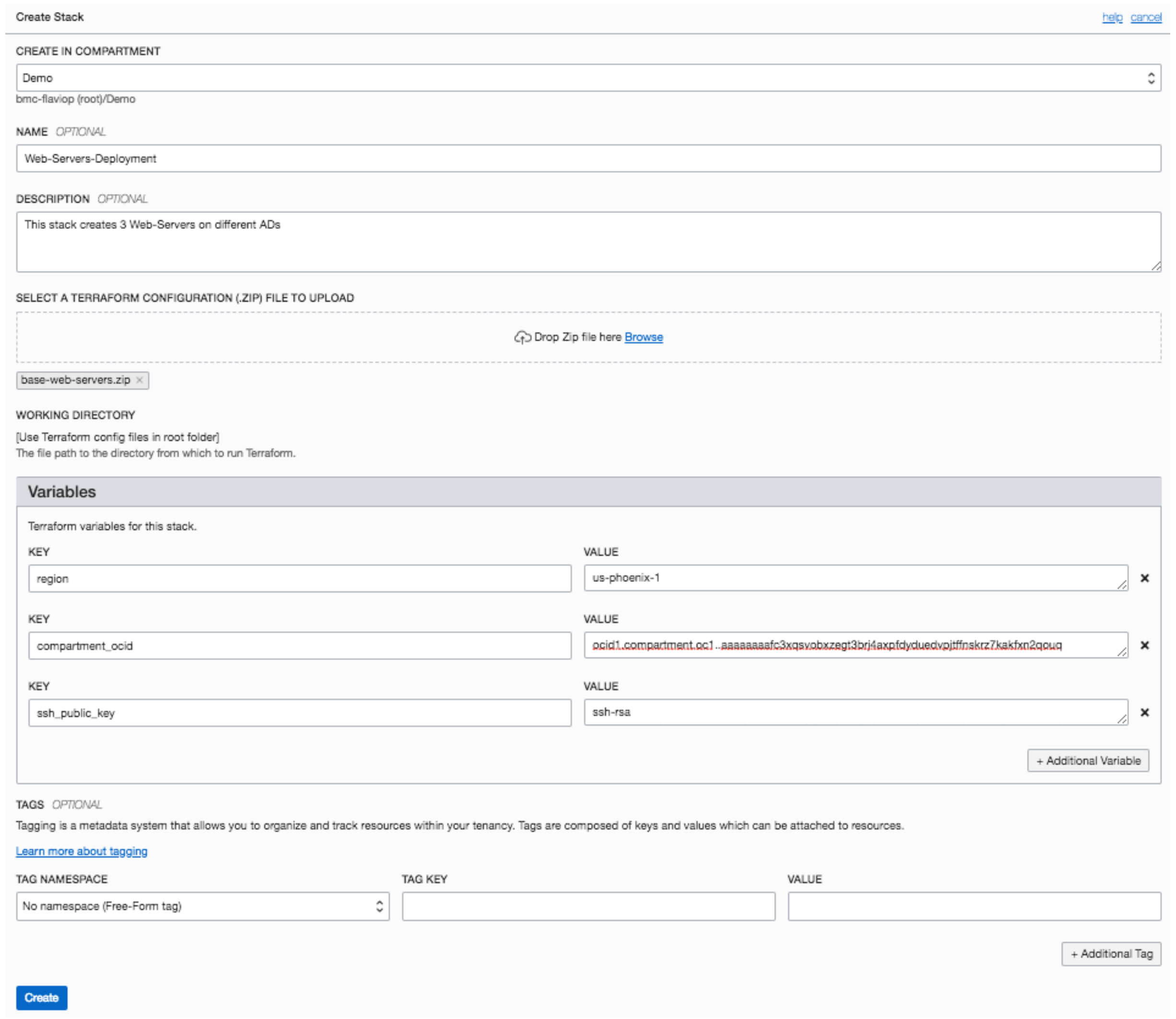


Resource Manager Workflow: Step 2

Create a Stack



- Stack represents a set of resources you manage within a compartment
- Each Stack maps to Terraform configuration files and a Terraform state file


Create Stack

CREATE IN COMPARTMENT

Demo

bmc-flaviop (root)/Demo

NAME *OPTIONAL*

Web-Servers-Deployment

DESCRIPTION *OPTIONAL*

This stack creates 3 Web-Servers on different ADs

SELECT A TERRAFORM CONFIGURATION (.ZIP) FILE TO UPLOAD

Drop Zip file here Browse

base-web-servers.zip ×

WORKING DIRECTORY

[Use Terraform config files in root folder]
The file path to the directory from which to run Terraform.

Variables

Terraform variables for this stack.

KEY	VALUE
region	us-phoenix-1
compartment_ocid	ocid1.compartment.oc1..aaaaaaaafc3xqsvobxzegt3brj4axpfdyduedvpjtffnskrz7kakfxn2qouq
ssh_public_key	ssh-rsa

TAGS *OPTIONAL*

Tagging is a metadata system that allows you to organize and track resources within your tenancy. Tags are composed of keys and values which can be attached to resources.

Learn more about tagging

TAG NAMESPACE	TAG KEY	VALUE
No namespace (Free-Form tag)		

Resource Manager Workflow: Step 3

Run a Terraform Job

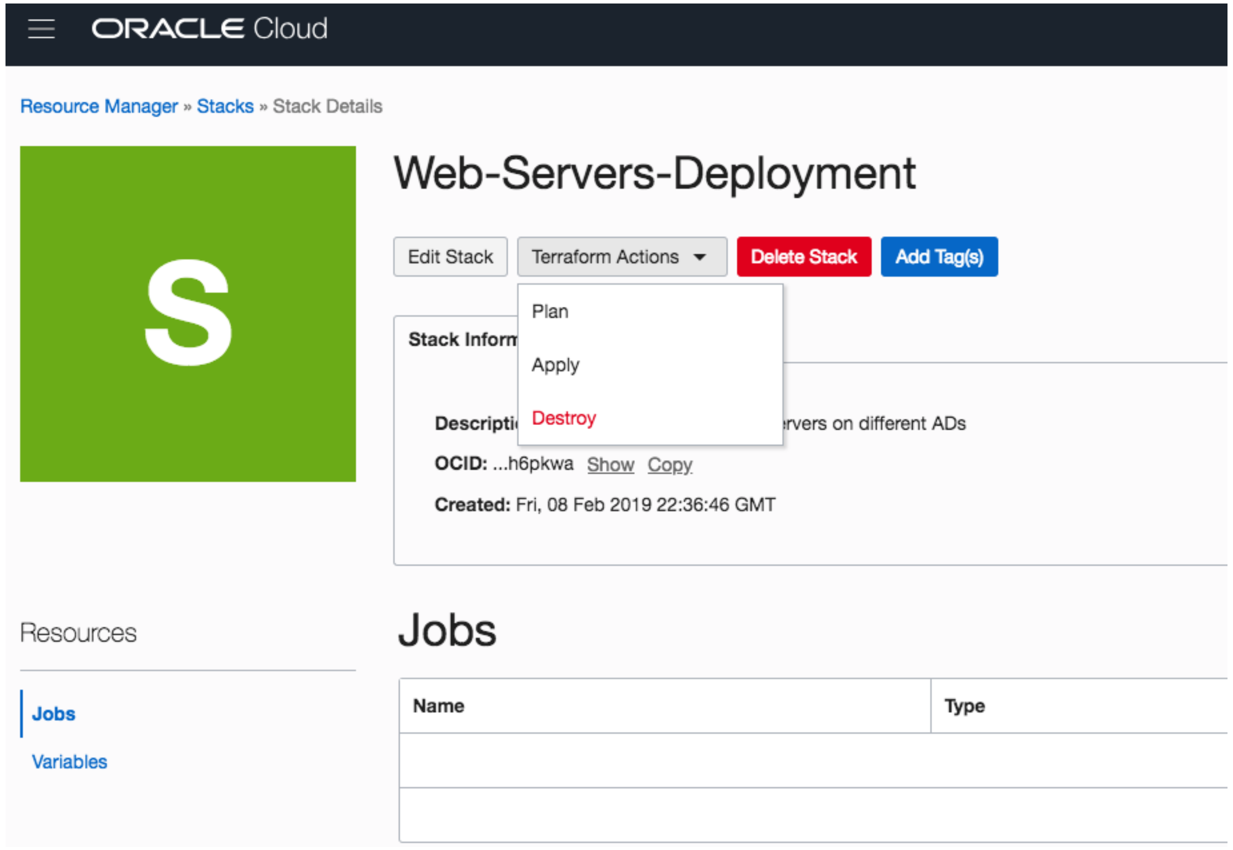
Terraform

Define configuration

Create a Stack

Run a Job

- A Job is a Terraform Action executed against a Stack
- Job actions include Plan, Apply, and Destroy



ORACLE Cloud

Resource Manager » Stacks » Stack Details

Web-Servers-Deployment

Edit Stack | Terraform Actions | Delete Stack | Add Tag(s)

Plan
Apply
Destroy

Stack Inform

Descripti ... rvers on different ADs

OCID: ...h6pkwa Show Copy

Created: Fri, 08 Feb 2019 22:36:46 GMT

Resources

Jobs

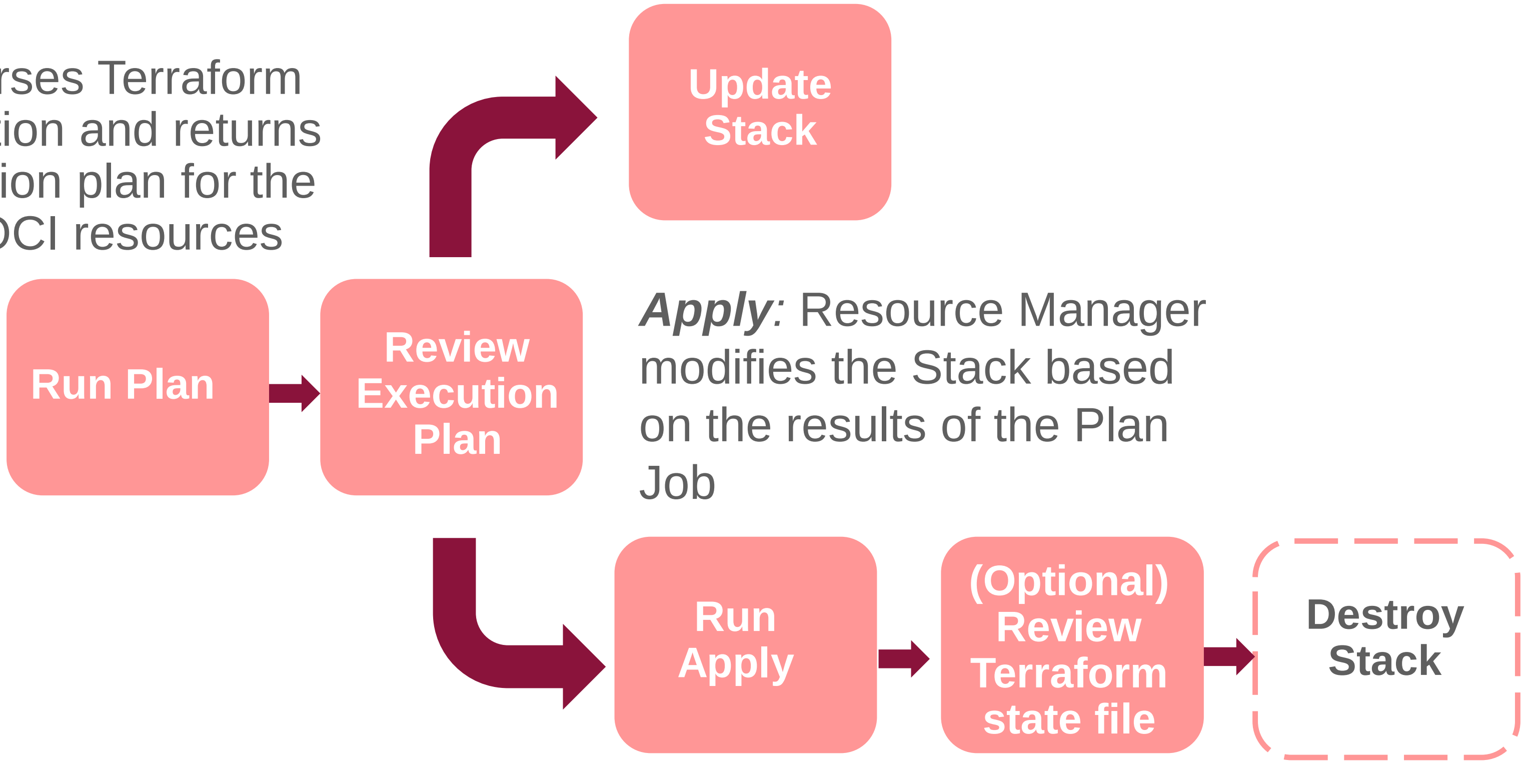
Variables

Jobs

Name	Type

Resource Manager Execution

Plan : Parses Terraform configuration and returns an execution plan for the effected OCI resources



Apply: Resource Manager modifies the Stack based on the results of the Plan Job

Destroy :Terraform deprovisions all of the resources in the Stack

Knowledge Check

What is the best practice recommendation for where to store your Infrastructure as Code?

A. On a USB thumb drive that you can easily share with co-workers.
B. On a network file server with tightly controlled LDAP permissions.
C. In a private source control repository like Git or Bitbucket.
D. Create an Oracle Database instance and configure the schema to support JSON document storage. Convert your Terraform configuration to JSON and insert into the database.

Knowledge Check

What is the best practice recommendation for where to store your Infrastructure as Code?

A. On a USB thumb drive that you can easily share with co-workers.

B. On a network file server with tightly controlled LDAP permissions.

C. In a private source control repository like Git or Bitbucket.

D. Create an Oracle Database instance and configure the schema to support JSON document storage. Convert your Terraform configuration to JSON and insert into the database.

Knowledge Check

You have previously deployed a 3-tier web application using Terraform and have just been asked to change the amount of storage allocated to one of the VM Database instances (scale storage up). After modifying your **.tf** file to reflect the requested change, which command would you run to apply the change?

A. terraform destroy

B. terraform apply -plan=scale_db_storage

C. terraform apply -target=oci_database_db_system.MyVmDb

D. terraform taint oci_database_db_system.MyVmDb

Knowledge Check

You have previously deployed a 3-tier web application using Terraform and have just been asked to change the amount of storage allocated to one of the VM Database instances (scale storage up). After modifying your **.tf** file to reflect the requested change, which command would you run to apply the change?

A. terraform destroy

B. terraform apply -plan=scale_db_storage

C. terraform apply -target=oci_database_db_system.MyVmDb

D. terraform taint oci_database_db_system.MyVmDb

ORACLE

Summary

- Understand the concept of Infrastructure as Code and the value of Terraform
- Discussed the core capabilities of Terraform
- Addressed challenges of working with Terraform at scale
- Learned about using Resource Manager to simplify collaborate efforts

ORACLE®
Cloud Infrastructure

cloud.oracle.com/iaas

cloud.oracle.com/tryit