**2**

**Infrastructure Automation and Management**

One of the primary ways in which cloud-based infrastructure differs from traditional infrastructure is in its use of APIs to provision and manage infrastructure. This means that the traditional hardware procurement and refresh cycle can now be replaced with a simple API call. This simple but powerful construct opens the door to significant advantages and optimization in infrastructure management. Users no longer have to preplan for infrastructure needs; instead, they can provision infrastructure just in time. This also unlocks cost savings in elastically scaling infrastructure based on metrics or other criteria. The advantages of these optimizations also cascade into modern application design practices that can programmatically scale infrastructure along with applications based on real-time need.

**One Set of APIs, Different Ways to Call Them**

As with any other cloud provider, Oracle Cloud Infrastructure (OCI) provides application programming interfaces (APIs) for all its infrastructure resources. This means that every single infrastructure resource that OCI provides can be created and managed using its APIs. On top of these APIs, OCI provides software development kits (SDKs) for various programming languages, which make the process of calling these APIs from your favorite programming language easy and enables you to create a new breed of software that can provision and manage hardware by itself. For example, if you have a Java or Python application that needs to create compute resources or set up other infrastructure, you can use the SDK to make the call to the OCI APIs from your language of choice.

Taking it one step further, domain-specific languages (DSLs) have been developed to make the process of interacting with these APIs easier. These DSLs and tools that build on top of these DSLs bring the power of cloud APIs to infrastructure professionals who have no application development background or experience. Two such popular tools are Terraform and Ansible. Terraform uses a DSL named HashiCorp Configuration Language (HCL) to describe infrastructure and manage it. Ansible describes infrastructure and configuration in YAML format. Regardless of the format and the tools/clients used, the APIs are the primary endpoints to the cloud platform. Users interacting with OCI using a browser-based UI, a terminal-based CLI, or even infrastructure-management tools such as Terraform are simply using various channels to call the underlying APIs. As tools and technologies evolve, OCI keeps adding new ways to make it easier to call the APIs and use OCI.

The examples and later chapters in the book make extensive use of Terraform for infrastructure management. This chapter covers Terraform basics and the managed services in OCI that use Terraform.

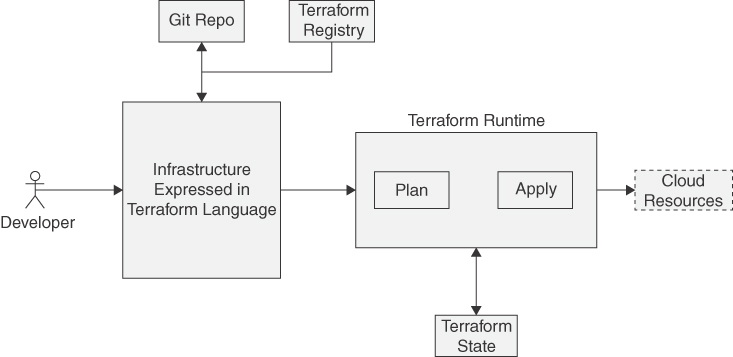
**A Quick Terraform Primer**

Terraform is an infrastructure-management tool from HashiCorp that helps you implement practices for managing infrastructure as code to create and manage reproducible infrastructure in the cloud and even across cloud providers. It works on the principle of defining infrastructure as it should exist (in a desired state) using a configuration language. This definition is then passed on to the Terraform tool, which makes appropriate cloud API calls to converge on the desired infrastructure state expressed in the configuration.

A user starts by defining the infrastructure topology and specifying how these resources are connected and configured using the Terraform language. This concise and human-readable configuration language forms the blueprint of the infrastructure that you want to create.

The Terraform language is expressed using the HashiCorp Configuration Language (HCL) syntax, which is also used in other HashiCorp products. This configuration file, often simply called the Terraform configuration, is typically source controlled to maintain a history of the changes to the desired infrastructure state. When Terraform configurations are paired with a source control system to store and track changes to the codified infrastructure, and a CI/CD tooling to manage the deployment of this infrastructure, you can quickly create a workflow that manages immutable, consistent, and repeatable environments. [Figure 2-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig01) illustrates the Terraform workflow.

Terraform interacts with external services such as cloud providers, Software as a Service (SaaS) services, and other APIs using plug-in modules called providers. A *provider* abstracts the cloud- or service-specific detail and models the resource mapping for the Terraform configuration language. Every Terraform provider adds a set of resource types and data sources that Terraform can use. You can use these resource types and data sources in the configuration to manage or query the respective cloud resources using Terraform. The Terraform Registry is the main repository of publicly available Terraform providers and hosts the OCI provider for Terraform.

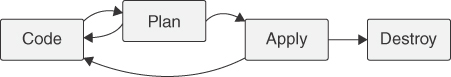


**Figure 2-1** Terraform Workflow

The Terraform Registry also acts as a repository for common cloud resource configurations that are designed to be reusable. These reusable configurations are called Terraform modules. Modules make it easy to ensure consistency and replicate organizational standards in your configurations. Modules can also cut down on duplicated resource definitions, making it easier to maintain these configurations over time. Consider an organization in which multiple teams are creating and using cloud resources such as networks. The organization might want to enforce specific configurations for all the networks that are created. When using a Terraform provider directly, users have full control over the resource parameters and may choose to configure that resource in any manner they choose. When using a Terraform module, however, many of the configuration options for the resource are already chosen by the module developer, and a minimal set of configuration options is presented to the user. In most cases, a module is functional in nature. This means that a module might have multiple resources, such as compute instances, load balancers, storage volumes, and more to quickly and consistently create a complete subsystem. Terraform modules make it easy for the user to create the resource with minimal input while adhering to the organizational standards.

When the desired configuration has been expressed in the Terraform language, the Terraform runtime can process that configuration to construct cloud API calls and execute those API calls against a cloud platform such as OCI. Because the configuration is expressed in terms of the resources and data sources supported by a provider, Terraform calls on the respective providers to make the actual API calls. Terraform configurations are required to declare the specific providers they use within the configuration, and Terraform initializes the providers when it runs. The same Terraform configuration can also use multiple providers, which makes it possible to interact with several services and cloud vendors within the same configuration to create and manage multicloud deployments. Terraform can validate the configuration, as well as do a dry run, by executing a terraform plan to see what impact running the configuration, executing a terraform apply, would have on the existing infrastructure.

[Figure 2-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig02) illustrates the typical cycle for managing infrastructure with Terraform. It starts with defining your infrastructure as code using the Terraform language. Next, you pass this configuration to Terraform and run a plan, which causes Terraform to inspect the existing resources and figure out the changes that need to be applied to converge the configuration to the one that has been provided. The result of the plan is the set of actions to be performed on cloud resources, such as what resources are to be created, modified, or deleted. If the plan looks valid and the actions that will be performed are acceptable, the developer can run an apply, which causes the plan to be applied. During the plan and apply operations, Terraform calls on the providers to make the API calls to query the existing resources or manage them on the remote cloud platforms. Resources can also be destroyed after their purpose has been fulfilled, which is common for ephemeral resources in the cloud. Using Terraform to destroy resources also ensures that no resource is overlooked or left behind because Terraform keeps tracks of every resource it creates. For instance, suppose that you write a Terraform configuration that creates a couple of compute instances and a load balancer that points to them. The first time this configuration is applied, all resources in the configuration are created. The configuration described in the Terraform files as code using the Terraform language now matches the resources created in the cloud. Now, if the developer updates the definition to include a third compute instance and updates the load-balancer definition to point at all three instances, Terraform will create a plan to add a new instance and to update the existing load balancer to include the newly created instance. Terraform will also figure out that it needs to create the instance first so that the instance’s IP address can be used as a back end for the load balancer to direct traffic. When you no longer need these resources, you can destroy them or scale down the resources. The code you created for the infrastructure can be run at any time to re-create the exact same resource configuration whenever you need it.



**Figure 2-2** The Typical Lifecycle for Managing Infrastructure with Terraform

This lifecycle can also address drift in your configurations. Drift is the change in configuration that is usually manually applied to your infrastructure, outside of what Terraform has been configured to do. For instance, consider a load balancer that is configured to listen on port 443. Now consider an operator that manually updates the configuration to open port 80 as well. This change is not described in the Terraform configuration; therefore, the next time Terraform runs, Terraform will detect this change and recommend reverting it to the original state by closing that port. This makes it easy to ensure that your infrastructure configurations are always well known, reproducible, and expressed as code that can be audited.

The power of implementing infrastructure as code is fully realized when you introduce a source control management system to store and track changes to the codified infrastructure and optional CI/CD tooling to manage the deployment of this infrastructure. You then have a complete workflow to create and manage immutable, consistent, and repeatable environments. No one needs to guess what configuration changes have happened to the infrastructure over time. Adopting this model means that the only way to manage the infrastructure is through Terraform, and Terraform is expressed as code with full version history and provenance tracked by the source control system.

This level of automation opens a new realm of possibilities for application development teams to become more agile than ever while optimizing cost. In a testing environment, for example, tests generally are not running continuously, which results in resource waste. With the help of automation tools such as Terraform and a CI/CD platform, application teams can create a test environment when required, run these tests, and destroy the environment after the tests have been completed. Because the infrastructure is described as code, it can consistently be re-created any number of times in any location. This portability also enables application teams to quickly create consistent environments across regions. These new opportunities help application teams expand quickly and consistently across the globe, create disaster recovery processes that are cost-optimized by running a scaled-down version of the infrastructure in the primary site, and more.

**A Basic Introduction to the Terraform Language**

Terraform configurations can be expressed using the Terraform language (using the HCL syntax) or using JavaScript Object Notation (JSON). Although expressing configurations in JSON makes it easier to parse using a variety of tools, the native Terraform language using the HCL syntax is more common and more expressive. Throughout this book, we use the more commonly used native syntax instead of JSON.

Terraform configurations written using the native language (HCL) are organized into blocks. *Blocks* are of different kinds and can represent an object such as a cloud resource, an output definition, a variable, or a provider configuration. [Listing 2-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_1) provides an example.

**Listing 2-1** Elements of the Terraform Language

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f023-01a)

# Block 1

variable "tenancy\_ocid" {

default = "xxxx.xxxx.xxxx.xxxx"

}

# Block 2. *Note that arguments for the provider definition have been truncated*

*for brevity*.

provider "oci" {

tenancy\_ocid = var.tenancy\_ocid

...

...

}

# Block 3

resource "oci\_core\_vcn" "my\_vcn" {

cidr\_block = "10.0.0.0/16"

...

...

}

# Block 4

resource "oci\_core\_subnet" "public" {

vcn\_id = oci\_core\_vcn.my\_vcn.id

cidr\_block = cidrsubnet(var.vcn\_cidr, 8, 0)

...

...

}

data "oci\_identity\_availability\_domains" "test\_availability\_domains" {

compartment\_id = var.tenancy\_ocid

}

Block 1 defines a *variable* named tenancy\_ocid and provides a default value for it using the argument default. This variable can be referenced from other parts of the code as var.tenancy\_ocid.

Block 2 declares and configures a *provider*. This block references the variable defined in the listing as var.tenancy\_ocid. The intention of the developer here is to build a configuration that can be run against multiple tenancies by turning the tenancy\_id into a variable that can be changed at runtime.

Block 3 is a *resource* block that represents a cloud resource. In this case, the oci\_core\_vcn resource models a virtual cloud network in OCI and is named my\_vcn. The CIDR block argument is provided with an explicit value.

Block 4 is also a *resource* block, and this one represents a subnet. Subnets are created inside (virtual) networks. In Block 4, the network to which this subnet should belong is identified by the vcn\_id argument. Note the value of this argument: oci\_core\_vcn.my\_vcn.id. It is clearly pointing to the VCN created in [Listing 2-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_1), identified by its resource type and name: oci\_core\_vcn.my\_vcn. However, the id attribute is not present; the id attribute is provided by the oci\_core\_vcn resource. The provider documentation lists the attributes that can be used to configure each resource type.[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_1a)

It is also worth pointing out that arguments can be updatable or non-updatable, depending on the behavior of the cloud resource. In the previous example, the vcn\_id argument for the resource is non-updatable. After the subnet is created, if the value is changed and Terraform is executed again, Terraform will notice the change. This effectively means that the subnet will be created under a different VCN. Terraform will delete the existing subnet and create a new subnet with the new arguments. Here, it makes sense because the updated configuration effectively means that a subnet is to be moved from one VCN to another and that the cloud provider does not support an “in-place” update to happen for this property of the resource. Another example could be updating the tags on a resource. Tags can be updated in place and thus will not trigger the re-creation of a resource. After the Terraform configuration to model the cloud resources has been created, you perform a dry run with Terraform. Performing a terraform plan shows the exact changes that will be effected. To make the changes that the dry run has outlined, a terraform apply can be executed.

**Terraform State Tracking**

Terraform tracks the state of your infrastructure using a .tfstate file. This state file is used to create associations between the cloud resources you create when you run the Terraform configuration and the entries defined in the configuration. In the previous example, when you create the actual subnet in OCI, the subnet’s identity (OCID) is associated with the definition "oci\_core\_subnet" "public". In the future, if the definition or configuration properties of the subnet change, Terraform will use this association to know which subnet in the VCN (identified by its OCID) to update or re-create the resource.

Before taking any action, Terraform runs a refresh to reconcile the state of the external real infrastructure resources with what is defined in the configuration. The state file is used for this reconciliation as well. If the external real infrastructure has been changed manually outside Terraform, there will be a delta between the state that Terraform has been tracking and the state that is on the real infrastructure. This *drift* detection capability in Terraform can greatly enhance the security posture by detecting unapproved infrastructure configuration changes, keeping the infrastructure in a well-known reproducible state. The state file also tracks dependencies between resources. In the previous example, if the VCN were to be re-created, the subnets would need to be re-created as well; Terraform automatically knows this because it tracks the dependencies between the resources in the state file.

By default, the Terraform state file is stored in a file named terraform.tfstate. For most real use, however, the default location is inadequate—especially when working in teams. Consider two developers running a Terraform configuration in parallel. They could be creating duplicate resources because they are using their own state files and are unaware of each other’s changes. Terraform supports remote state to solve this problem and offers several options to configure and store state remotely. Aside from making the state changes transparent to everyone in the team, remote state storage can also use locking. Locking the state protects teams from performing multiple simultaneous Terraform executions at the same time, thus ensuring that each Terraform run begins with the most recent updated state. Terraform can use one of several back ends to manage remote state; each back end provides support for a different storage system.

OCI object storage can be used as a remote state back end in Terraform. This allows Terraform to use the OCI object storage as the storage back end for Terraform state, making it consistent across all users of the Terraform configuration. Terraform can leverage OCI Object Storage using the HTTP back end or the S3 back end. [Table 2-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02tab01) compares the two storage options.

**Table 2-1** .tfstate Storage Option Comparison

| **HTTP Back End** | **S3 Remote State Back End** |
| --- | --- |
| Uses standard HTTP methods | Uses Object Storage S3-compatible APIs |
| Requires a preauthenticated request and no additional authentication | Requires additional per-user authentication |

**Note**

Terraform state files can potentially contain sensitive data, especially if you use Terraform to manage access keys, passwords, or cryptographic keys. Even without such data, the state file contains a complete blueprint for your cloud infrastructure and its topology. For this reason, it’s always preferable to consider the state file as a sensitive document.

**The OCI Terraform Provider**

The OCI Terraform provider[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_2a) is the component that connects Terraform to OCI services. It models the OCI services and APIs as Terraform objects such as resources and data sources that can be used in Terraform configurations. The OCI provider is open source and is available through the Terraform Registry, which hosts the providers for all major platforms and services. Developers and DevOps engineers can use the OCI Terraform provider to manage OCI resources wherever you use a Terraform distribution, including with Terraform Cloud and the OCI Resource Manager. The Terraform runtime and the OCI provider are also available for installation through Oracle’s public YUM repositories.

**Setting Up the OCI Terraform Provider**

When installing on OCI compute instances, the quickest way to install Terraform CLI and the OCI provider is to use the YUM repositories. The tools are included in the developer repository, which is usually disabled by default. Depending on the version of Oracle Linux, you can enable the repository and install the packages with the following commands:

For Oracle Linux 8:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f026-01a)

sudo yum-config-manager --enable ol8\_developer &&\

sudo yum install terraform

For Oracle Linux 7:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f027-01a)

sudo yum-config-manager --enable ol7\_developer &&\

sudo yum install terraform

Optionally, you can install the OCI Terraform provider using YUM, although it might be preferable to manage versions of the provider for each Terraform configuration basis by letting Terraform download and manage the Terraform provider as part of initialization for each configuration. The Terraform provider for OCI is available in the Oracle YUM repositories with the package name terraform-provider-oci and can be installed with the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f027-02a)

sudo yum install terraform-provider-oci

**Note**

If you work in a highly sensitive environment, you might want to consider the FIPS-compatible version of the OCI provider. The FIPS-compatible version of the provider ensures that traffic from Terraform to OCI service API endpoints transits over a TLS connection established with an HTTP client using FIPS-certified encryption. The FIPS version of the OCI Terraform provider uses the FIPS 140-2 certified Oracle Cloud Infrastructure Cryptographic Library for Kubernetes instead of the Go native cryptography implementation. Install the FIPS-compatible provider as follows

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f027-03a)

sudo yum-config-manager --enable ol8\_developer &&\

sudo yum install terraform &&\

sudo yum install terraform-provider-oci-fips

To install Terraform CLI manually on your workstation, you can download it directly from HashiCorp (<https://www.terraform.io/downloads>) for your specific operating system.

After Terraform is installed, when it encounters a configuration that uses the OCI provider in a provider block, Terraform downloads the provider as part of the terraform init process. You can also pin your configuration to a specific version of the provider. This is useful when you want to ensure that you always use a provider version that you have tested your configuration with or when you need to maintain software versioning for compliance reasons. When you always use the latest version of a provider, you can potentially get unexpected behavior if a provider makes a non-backward-compatible change. You can pin the provider version by specifying the version in the required\_providers.oci.version argument, as shown in [Listing 2-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_2).

**Listing 2-2** Configuring the OCI Provider

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f027-04a)

terraform {

required\_providers {

oci = {

source = "hashicorp/oci"

version = ">= 4.50.0"

}

}

}

provider "oci" {

# variables are not shown

region = var.region

tenancy\_ocid = var.tenancy\_ocid

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

The configuration in [Listing 2-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_2) shows that the version of the provider required is set as version 4.50.0 or later. You can also see that the provider has been initialized with a set of provider configuration arguments that include the following:

* region**:** The OCI region where this configuration will be applied
* tenancy\_ocid**:** The tenancy OCID against which this configuration will be applied
* user\_ocid, fingerprint, private\_key\_path**:** API signing key credentials for authenticating with OCI

Using API keys is the default and most common method for authenticating Terraform with OCI. If you have the OCI CLI installed,[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_3a) the Terraform provider can optionally use the OCI CLI configuration file for authentication. This can be handy because the OCI CLI configuration file supports profiles to work with multiple OCI tenancies or to use different identities to authenticate with OCI, enabling you to switch between identities and tenancies while externalizing the authentication information from your Terraform code. You can configure the provider to use the OCI CLI configuration, as demonstrated in [Listing 2-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_3). The default configuration filename and location is ~/.oci/config. You change this by setting the environment variable OCI\_CLI\_CONFIG\_FILE to point to a CLI configuration file at an arbitrary location.

**Listing 2-3** Using the CLI Configuration to Provide Authentication Information

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f028-01a)

provider "oci" {

tenancy\_ocid = var.tenancy\_ocid

config\_file\_profile= 'profile\_name\_in\_CLI\_configuration\_file'

}

**Authenticating Without API Keys**

When using automation systems and CI/CD tools, using an API key might not be desirable. In these cases, you can consider *Instance Principal*–based authentication. This authentication method is applicable only if Terraform is being executed from an OCI compute instance. This relies on Instance Principals in OCI and the policies that govern them. When using Instance Principals, you do not need to provide the arguments used for API key-based authentication. Your OCI Provider configuration could look like [Listing 2-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_4).

**Listing 2-4** Using Instance Principals to Provide Authentication Information

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f029-01a)

provider "oci" {

auth = "InstancePrincipal"

region = "${var.region}"

}

You can also use the token-based authentication driven by the OCI CLI. This method avoids the use of an API signing key and provides an interactive way to authenticate users. This authentication method uses the OCI CLI *session authenticate* flow that uses an OAuth implicit grant flow to authorize Terraform.

**Managing OCI Resources with Terraform**

When you have Terraform installed and configured, you can start creating Terraform configurations for OCI. The OCI Terraform provider defines several resources and data sources that enable you to interact with the cloud resources offered by OCI. *Resource* blocks in Terraform represent a cloud resource such as a compute instance or a load balancer. *Data sources* provide data about cloud resources. They are typically used to *query* a cloud provider to get data about one or more resources so that the Terraform code can make choices based on that data.

To understand this, consider a real scenario. You need to create two compute instances, one that uses an ARM-based CPU and another that uses an Intel/AMD-based CPU. These are different CPU architectures and require different versions of the operating systems to be installed on them. [Listing 2-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_5) is a Terraform code snippet that makes the instance shape a variable. If the user chooses an instance based on the ARM architecture, the OS image is different than if the user had chosen a shape that uses the x86 architecture.

**Listing 2-5** Using Data Source to Create More Dynamic Configurations

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f029-02a)

# Only relevant attributes are shown below.

resource "oci\_core\_instance" "RStudio" {

display\_name = "MyInstance"

shape = var.instance\_shape

source\_details {

source\_type = "image"

source\_id = data.oci\_core\_images.InstanceImageOCID.images[0].id

}

}

data "oci\_core\_images" "InstanceImageOCID" {

compartment\_id = var.compartment\_ocid

operating\_system = var.instance\_os

operating\_system\_version = var.linux\_os\_version

shape = var.instance\_shape

}

variable "instance\_shape" {

description = "Instance shape"

}

variable "instance\_os" {

description = "Operating system."

default = "Oracle Linux"

}

variable "linux\_os\_version" {

description = "Operating system version."

default = "7.9"

}

To use the latest version of the operating system image available for each architecture, you can use a data source. The set of parameters and filters in a data source lets you identify a resource (in this case, the operating system image that satisfied your conditions) without having to hard-code anything. The data source for the images accepts the shape as an argument and lists only images that are compatible with the provided shape. This data obtained by the data source can be used by the compute instance resource so that the instance uses a compatible operating system image.

The OCI Terraform provider defines resources and data sources for almost all services that are available through the OCI API.

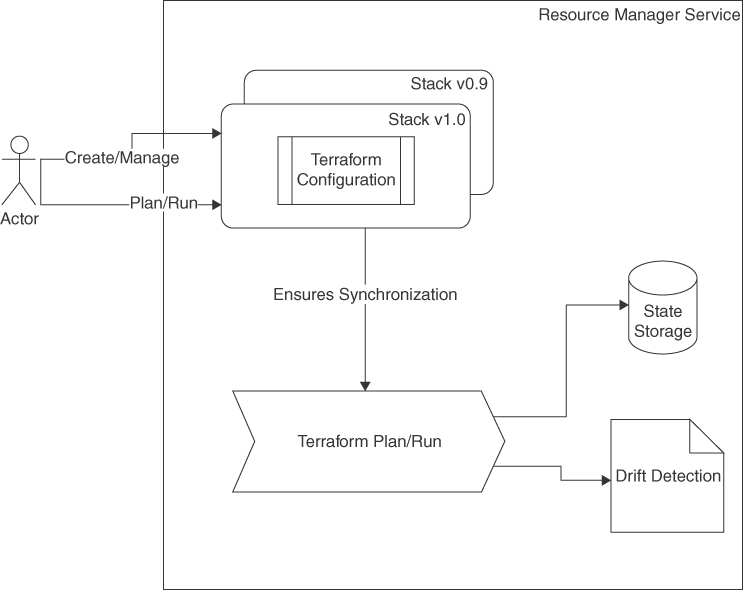
**Simplifying Infrastructure Management with the Resource Manager Service**

You already saw how OCI resources can be managed through infrastructure-management tools such as Terraform. The OCI provider for Terraform allows Terraform to hook into the OCI APIs and manage OCI resources using a Terraform language. You also learned how Terraform itself manages state and how to use OCI object storage for shared state when using teams. These approaches still assume that the developers have the tooling installed and configured. Large teams can still run into issues when trying to standardize Terraform versions and their execution environments and trying to coordinate between developers, despite having state file locking.

To address these shortcomings, OCI offers a fully managed Terraform platform called Resource Manager Service. The Resource Manager Service is a managed Terraform platform that can manage Terraform configurations as *stacks* that encapsulate multiple resource definitions. Stacks are first-class OCI resources that enable teams to coordinate their infrastructure management activity. Stacks can be based on Git repositories, ZIP files, or Terraform configuration files uploaded to the Resource Manager Service.

Resource Manager keeps track of the Terraform configuration and provides lifecycle management, as in managing the execution of the plan and apply stages. As a managed service, it provides the execution environment for developers to run Terraform configurations, removing the need to maintain local tooling. Developers can also run multiple Terraform configurations in parallel using Resource Manager. Additionally, Resource Manager keeps track of the Terraform state file, enabling team-based development. The Resource Manager Service manages the Terraform execution environment and the execution process, enabling coordinated infrastructure management for teams. [Figure 2-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig03) shows the various aspects of the Resource Manager Service.

Stacks also enable the developer to optionally omit certain Terraform provider configuration parameters, such as the user\_ocid, from the Terraform definition. This is because stacks themselves are OCI resources that an authenticated and authorized user is accessing, and the service can fill in some of the provider initialization parameters from the execution context. This makes stacks more portable because they can use their execution environment to infer these values. Take a look at the example in [Listing 2-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_6).



**Figure 2-3** Resource Manager Service Components

**Listing 2-6** OCI Provider Configuration with Terraform

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f032-01a)

provider "oci" {

# variables are not shown

region = var.region

tenancy\_ocid = var.tenancy\_ocid

# Authentication parameters

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

The same configuration can be expressed in Resource Manager, as shown in [Listing 2-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_7).

**Listing 2-7** OCI Provider Configuration When Using Resource Manager Service

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f033-01a)

provider "oci" {

# variables are not shown

tenancy\_ocid = var.tenancy\_ocid

region = var.region

# The authentication parameters are not required since the Resource Manager

authenticates the user.

}

As you can see, the authentication parameters have been omitted from the configuration. This is possible because the omitted values can be inferred by the service and passed on to the Terraform runtime.

Resource Manager not only manages Terraform executions and lifecycle, but it also provides valuable additional integration with other OCI services and features. These include generating Resource Manager stacks from resource creation wizards, resource discovery, and drift detection. The following sections cover these concepts in more detail.

**Helm and Kubernetes Providers**

As a managed platform, Resource Manager Service manages the Terraform providers that can be used through Resource Manager. Apart from the OCI provider, several popular third-party providers are supported, including Ansible, Vault, TLS, Kubernetes, and Helm. In later chapters, you interact with both the Helm and Kubernetes providers for Terraform.

The Kubernetes provider for Terraform adds support for creating and managing Kubernetes objects as Terraform resources. Creation, modification, and deletion of resources produces the same effect on the Kubernetes objects. The provider makes it possible to manage Kubernetes objects in a Kubernetes cluster, much the same as it enables you to manage infrastructure resources on a cloud provider. The example in [Listing 2-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_8) shows a PodSpec as a Terraform resource. The kubernetes\_pod resource is provided by the Kubernetes provider for Terraform and models a standard Kubernetes PodSpec.

**Listing 2-8** Managing Kubernetes Resources Through Terraform

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f034-01a)

resource "kubernetes\_pod" "test" {

metadata {

name = "nginx"

}

spec {

container {

image = "nginx:1.7.9"

name = "example"

port {

container\_port = 8080

}

liveness\_probe {

http\_get {

path = "/nginx\_status"

port = 8080

}

initial\_delay\_seconds = 3

period\_seconds = 3

}

}

}

}

[Listing 2-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_9) shows the equivalent PodSpec.

**Listing 2-9** Managing Kubernetes Resources Through PodSpec

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f034-02a)

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 8080

livenessProbe:

httpGet:

path: /nginx\_status

port: 8080

initialDelaySeconds: 3

periodSeconds: 3

Similar to the Kubernetes provider, but perhaps more useful in the context of automation, is the Helm provider. Helm itself is a way to package, deploy, and manage complex application deployments on Kubernetes. A set of Kubernetes objects such as pods, services, and config-maps that together make up an application are parametrized and managed as a Helm release. The example in [Listing 2-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_10) shows how a packaged Helm chart, the template for generating Kubernetes manifests that make up the application, is defined as a Terraform resource. Creation of the resource implies creation of the release—or, in other words, deployment of the application Helm chart.

**Listing 2-10** Managing Helm Releases Using Terraform

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f035-01a)

resource "helm\_release" "nginx\_ingress" {

name = "nginx-ingress-controller"

repository = "oci://ghcr.io/nginxinc/charts"

chart = "nginx-ingress"

version = "0.18.0"

set {

name = "service.type"

value = "ClusterIP"

}

}

The equivalent Helm command follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f035-02a)

helm install nginx-ingress-controller oci://ghcr.io/nginxinc/charts/nginx-

ingress --version 0.18.0 –set service.type=ClusterIP

The support for the Helm and Kubernetes providers makes it possible for developers to create solutions that automate not just infrastructure provisioning, but also the deployment and management of cloud native workloads as resources managed through Terraform. The Kubernetes best practice at the time of this writing is to keep the infrastructure components and Kubernetes/Helm provider-managed resources (Kubernetes objects and Helm charts) as separate Terraform modules.

The complete list of supported providers are documented[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_4a) and are frequently updated as new providers are added.

**Generating Resource Manager Stacks**

Resource Manager is integrated with other OCI resource creation wizards in the OCI console to make the process of generating resource definitions easy. A good example is the Oracle Container Engine for Kubernetes (OKE). OKE is a service that requires multiple supporting resources, such as networks, and offers many configuration options when you create a cluster. Instead of creating the cluster manually, which makes the cluster difficult to reproduce because of the sheer number of available options, you can use Terraform to encapsulate all the options and configuration, which can be then managed as a Resource Manager stack. However, if you’re not familiar with the Terraform language, the cluster re-creation workflow in the console conveniently provides a way to export the configuration of a cluster as a Resource Manager stack after the input is gathered. Thus, you can provide all the configuration options for the cluster to the cluster re-creation wizard, which then exports that configuration to a Resource Manager stack. The stack can then be executed, creating an OKE cluster along with all the required components, such as VCNs, security lists, node pools, nodes, placement configurations, and so on.

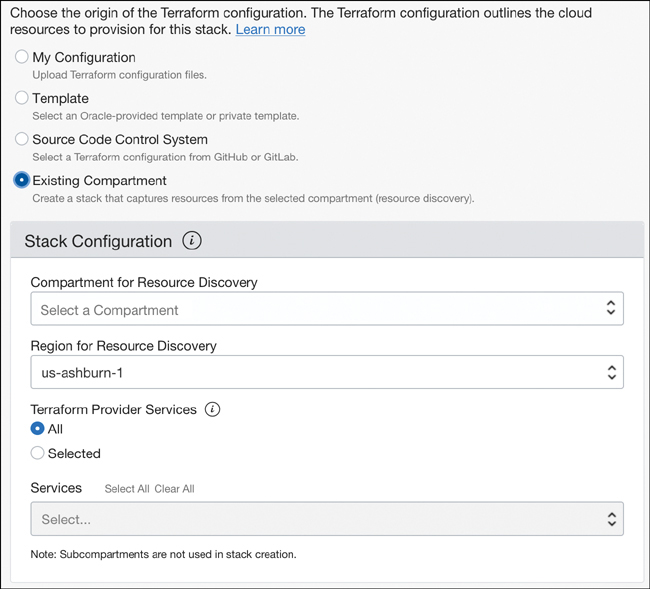
**Resource Discovery**

If you already have a hand-crafted solution created in OCI using a set of resources, the resource discovery feature in Resource Manager can generate a Terraform configuration out of these existing resources to make the solution more portable and easier to replicate across environments and regions. Resource discovery works at the compartment level; the compartment provided to the service is considered the source compartment. Resource discovery generates a Resource Manager stack along with a Terraform state file that represents all the supported resources that belong to the given source compartment. The discovery process does not descend into nested compartments or support multiple source compartments for the same stack.

It is also important to note that, in most cases, the stacks generated by resource discovery provide a starting point for the code representing the resources and require slight modifications to be run. For instance, sensitive information such as passwords used when creating a database are not included in the generated code, for obvious security reasons. Resource attributes whose values are omitted from the generated stack are placed in an ignore\_changes meta-argument in a nested lifecycle block for the resource definition. Similarly, resources that have been terminated or are otherwise inactive are generally excluded from the generated stack. [Figure 2-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig04) shows the option for creating a stack that captures resources from a specified compartment.

**Note**

lifecycle is a nested block for meta-arguments in Terraform that can appear in any resource block. The ignore\_changes meta-argument identifies a list of attributes that may change outside Terraform after the resource is created. This signals to Terraform that changes to these values are acceptable and do not warrant an update to reset the values. In the case of resource discovery, attributes without values are placed in the ignore\_changes list to ensure that a Terraform plan will still run without failure. When creating the resources, the developer should provide appropriate values because many of these, such as passwords, could be mandatory parameters for resource creation. Where appropriate, the developer can also move them out of the ignore\_changes list.



**Figure 2-4** Resource Discovery in the Resource Manager Service

**Drift Detection**

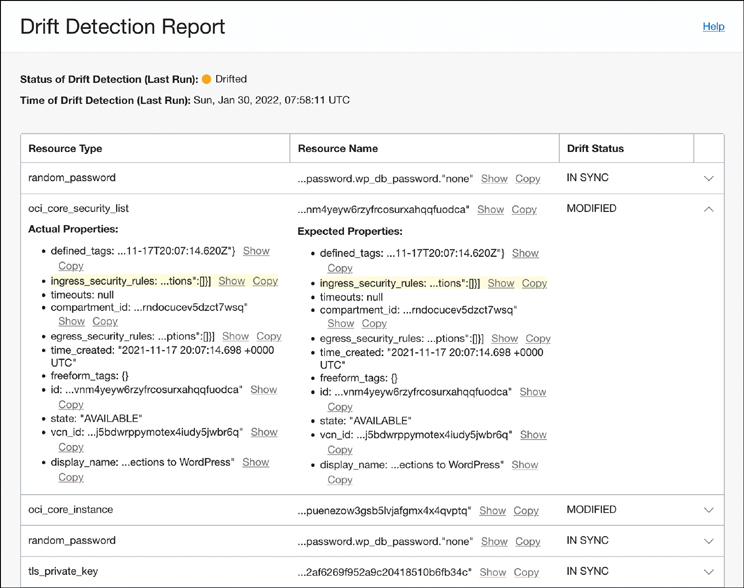
A critical feature that the OCI Resource Manager provides to streamline infrastructure management at scale is *drift detection*. As the name suggests, this feature can compare actual infrastructure and its parameters with the expected infrastructure configuration to identify drift or deviations.

As your cloud-based infrastructure scales, a compelling motivator will be moving toward an elastic model to optimize cost and infrastructure usage with infrastructure-as-code practices. Imagine that you’re managing your infrastructure through code. You expect your infrastructure to be in sync with that code, which is fundamental to realizing immutable infrastructure. Infrastructure automation makes it easy to implement immutable infrastructure, but it does not prevent users with access to the infrastructure from making ad-hoc changes to it, such as opening a port on your network security list. A fully mature practitioner will have multiple checks and balances and will have well-defined supporting processes such as security policies to keep infrastructure secure and immutable. Even in these cases, a malicious user or attacker could try to make changes to the infrastructure and compromise it.

Periodically verifying the infrastructure against the configuration defined in the code therefore provides a checkpoint that the infrastructure is still compliant with the definitions. A deviation could point to ad-hoc changes, configuration that is not yet captured in the code, or, in the worst case, a potential intrusion or attack on your infrastructure. The drift detection feature in Resource Manager performs this validation by comparing the current state of the resources with the last executed state for the stack. Drift detection produces a report of the drift that shows the actual resources and their parameters compared to the configuration in the stack. Changes are highlighted to make it easier to identify the changes, and the stack is marked as *Drifted* if drift is detected. Drift detection reports are run as asynchronous work requests, and their progress can be tracked through these work requests. The latest drift detection report is available on the stack details page or through the More Actions menu; historical reports are available from their work requests. [Figure 2-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig05) shows a drift detection report highlighting how the actual configuration has drifted from the expected configuration.

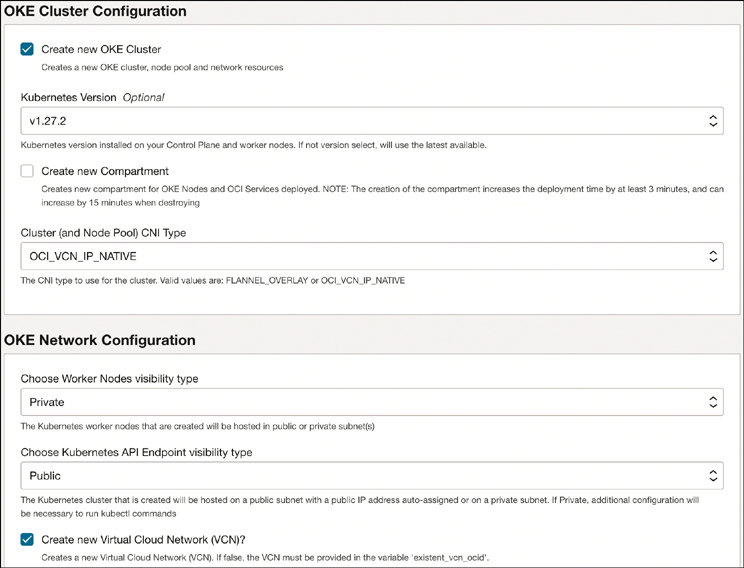
**Generating a User Interface from Terraform Configurations with a Custom Schema**[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_5a)

The Resource Manager Service goes beyond just managing the lifecycle for Terraform configurations. It can also extend the OCI console UI to make Terraform execution more approachable to end users. This enables developers to create Terraform configuration-based solutions that can be configured and deployed by end users who have no Terraform experience with an intuitive UI.



**Figure 2-5** Drift Detection Report Highlighting Changes to Resources

When a stack is loaded, the Resource Manager Service inspects the Terraform configuration in it to identify the variable declarations and other user input required for the stack to execute. On the standard command line, Terraform prompts the user to provide input through the terminal. Resource Manager, on the other hand, renders a text input box on the console UI for the user to provide variable values. Although this is perfectly acceptable in some cases, developers can include an optional schema file along with a stack. The schema file describes how the variables and outputs of the terraform configuration look and behave. [Figure 2-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig06) shows how Resource Manager can render Terraform variables as intuitive onscreen controls.



**Figure 2-6** Resource Manager Rendering Terraform Variables with the Help of UI Hints Provided in the Schema File

The schema file should be written in YAML and included at the root of the stack. The YAML document contains UI hints and structure that help the Resource Manager service to extend the OCI console UI to support much more complex input fields, queries, and validation. Each schema document defines keys for various purposes. The values can be of various types, such as *string*, *numeric*, *enum*, types that query OCI APIs for existing resources, or an expression that can be evaluated into values.

[Listing 2-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_11) provides an example of a schema file.

**Listing 2-11** Example Schema File for Terraform Stacks

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f040-01a)

# Stack Metadata

title: "MuShop Cloud Native App"

description: "Microservices demo App for Oracle's Container Engine for Kubernetes

(OKE)"

schemaVersion: 1.1.0

version: "20190304"

# Variable Groups

variableGroups:

- title: "General Configuration"

variables:

- app\_name

- password

- show\_advanced

visible: true

outputs:

service\_ip:

type: link

title: Application Login

description: Open the login page for the application

primaryOutputButton: service\_ip

# Variables

variables:

app\_name:

type: string

title: "Cluster Name Prefix"

required: true

visible:

and:

- create\_new\_oke\_cluster

app\_password:

type: password

required: true

title: Admin Password

description: The password for the admin user

pattern: "^(?=.\*[!@#%^\*\_+\\-:?.,\\[\\]\\{\\}])(?=.\*[0-9])(?=.\*[a-z])(?=.\*

[A-Z])(?!.\*[$\\(\\)]).{8,32}$"

visible: true

node\_pool\_shape:

type: oci:core:instanceshape:name

title: "Select a shape for the Worker Nodes instances"

required: true

dependsOn:

compartmentId: compartment\_ocid

visible:

and:

- create\_new\_oke\_cluster

A typical schema file like the one in the example is structured into at least four sections:

* Stack metadata.
* Variable groups: These are denoted by variableGroups.
* Output section: Output groups can group outputs into various sections for display after the stack has been applied.
* Variable metadata: The variables key specifies the display properties and behavior of variables in the Terraform configuration.

The next pages look at each section in detail.

**Stack Metadata**

The stacked metadata includes information about the stack itself. It uses keys such as title, description, version, and logoUrl that can provide metadata about a stack, including its purpose and publisher information, along with how to display the stack in the OCI console. The metadata for the stack is shown on the console UI at the time the stack is loaded and provides a description of the functionality that the stack provides. [Listing 2-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_12) shows an example.

**Listing 2-12** Stack Metadata in a Schema File

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f042-01a)

title: "MuShop Cloud Native App"

description: "Microservices demo App for Oracle's Container Engine for Kubernetes

(OKE)"

schemaVersion: 1.1.0

version: "20190304"

locale: "en"

logoUrl: <URL/URL encoded data>

**Variable Groups**

This element groups variables into various sections for display on the UI, such as grouping all networking parameters together. The variable groups are listed under the key variableGroups, which is at the root of the document. The example in [Listing 2-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_13) has three separate variable groups, called General Configuration, OKE Cluster Configuration, and OKE Worker Nodes. Each variable group has a title, which forms the title for the section of input fields in the UI, and a list of variables. The list of variables is a reference to the variable definitions that are listed further down in the document. Each of the variable groups can also have a visible attribute that can be evaluated to a Boolean that toggles the display of the entire section.

**Listing 2-13** Variable Groups Can Help Structure Related Terraform Variables on the User Interface

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f043-01a)

variableGroups:

- title: "General Configuration"

variables:

- app\_name

- password

- show\_advanced

visible: true

- title: "OKE Cluster Configuration"

variables:

- create\_new\_oke\_cluster

- existent\_oke\_cluster\_compartment\_ocid

- existent\_oke\_cluster\_id

- k8s\_version

- title: "OKE Worker Nodes"

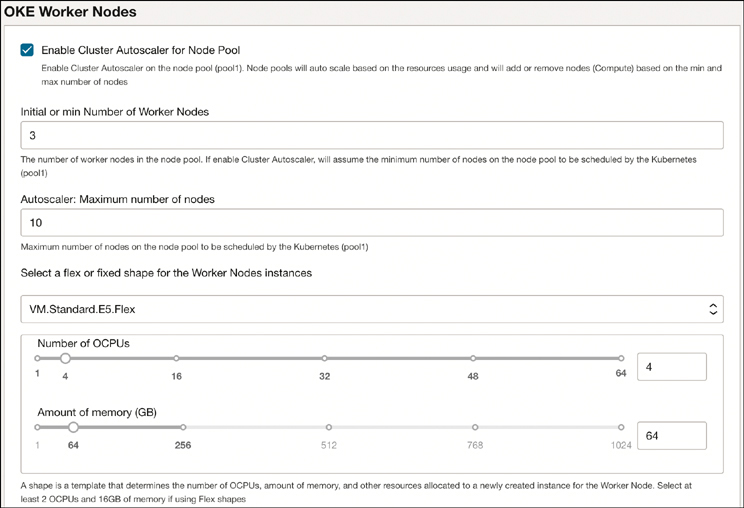
variables:

- num\_pool\_workers

- node\_pool\_shape

- node\_pool\_name

In the example, the first variable group is the General Configuration. Here you can see that this group has two keys defined, the title and variables. The section is presented to the user under a section with the title specified here, and the section will render the variables listed in the variables list. The display of the individual variables listed—app\_name, password, and show\_advanced—depends on these variables’ metadata defined in the variables section at the root of the document. [Figure 2-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig07) shows the OKE Worker Nodes variable group and demonstrates how a variable group is rendered.



**Figure 2-7** OKE Worker Nodes Variable Group

**Output Section**

The variable group section is followed by the output section. Here, we see two separate outputs defined. Outputs from the stack are displayed on the Application Information tab on the stack page after the stack has been run at least once. [Listing 2-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_14) shows an example.

**Listing 2-14** The Output Section Provides Structure to Terraform Outputs

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f044-01a)

outputs:

service\_ip:

type: link

title: Application Login

description: Open the login page for the application

generated\_ssh\_private\_key:

title: Generated Private Key

description: The auto-generated private key. Save this to a file to SSH into

the instance.

type: copyableString

visible: true

The first output in the example is called the service\_ip and has attributes that include type, title, and description. The title and description provide the metadata for displaying the value of the output on the UI. The type determines how the value itself is rendered; here, it is rendered as a hyperlink. The other output is named generated\_ssh\_private\_key and is of type copyableString. When rendering the value, it will be shortened for better display, along with Show and Copy buttons for users to easily see or copy the value.

You also see a primary output button. The Application Information tab can optionally render a button on the Information area, which is determined by the primaryOutputButton. In this example, it references the service\_ip, which means that the service\_ip will be rendered as the primary output button on the Application Information tab. The button text uses the title for the service\_ip; clicking the button opens the hyperlink that is the value for the service\_ip.

**Variable Definitions**

The variables used in the Variable Groups are defined here, and these correspond to the variables declared in the Terraform configuration. Variable definitions provide metadata for the variables, such as their type. Types such as numeric, string, and password affect both how the variable is rendered on the UI and its behavior. The attributes of a variable definition can include expressions that depend on other attributes as well. For instance, it is possible to specify that the app name should be visible if the create\_new\_oke\_cluster variable is true, as shown in [Listing 2-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_15).

**Listing 2-15** Variable Definitions Provide Metadata to Influence the Look and Behavior for Terraform Variables

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f045-01a)

app\_name:

type: string

title: "Cluster Name Prefix"

required: true

visible:

and:

- create\_new\_oke\_cluster

The definition of the app\_password variable in [Listing 2-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_16) specifies that it is of type password, which provides a hint to the Resource Manager to render it as a password field on the UI. It is also marked as a required variable, which means that a value is expected from the user; the UI will display an error if the user does not provide one. For this variable, the title and description give the UI hints to display this variable onscreen. The title provides the prompt in the description that supplies a small textual description under the input field. The pattern is a regular expression to validate the text input on this field. If the value provided by the user does not match this regular expression, the service rejects the value and an error is displayed onscreen identifying the field that failed the validation.

**Listing 2-16** Variable Definitions Can Validate Input Data

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f046-01a)

app\_password:

type: password

required: true

title: Admin Password

description: The password for the admin user

pattern: "^(?=.\*[!@#%^\*\_+\\-:?.,\\[\\]\\{\\}])(?=.\*[0-9])(?=.\*[a-z])(?=.\*[A-Z])

(?!.\*[$\\(\\)]).{8,32}$"

visible: true

Resource Manager supports several other types of variables, including Booleans, OCI resources, and enumerations. [Listing 2-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_17) shows the usage of some of these types and how they can interact with each other.

**Listing 2-17** Variable Definitions Can Reference Other Variables to Create Complex UI Behavior

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f046-02a)

show\_advanced:

type: boolean

title: "Show advanced options?"

description: "Shows advanced options, such as customer-managed encryption keys"

visible: true

create\_new\_oke\_cluster:

type: boolean

title: "Create new OKE Cluster"

existent\_oke\_cluster\_compartment\_ocid:

type: oci:identity:compartment:id

title: "Existent OKE Cluster Compartment"

description: "The compartment where you find the existent OKE Cluster"

default: compartment\_ocid

required: true

visible:

not:

- create\_new\_oke\_cluster

existent\_oke\_cluster\_id:

type: oci:container:cluster:id

title: "Existent OKE Cluster"

required: true

dependsOn:

compartmentId: existent\_oke\_cluster\_compartment\_ocid

visible:

not:

- create\_new\_oke\_cluster

k8s\_version:

type: oci:kubernetes:versions:id

dependsOn:

compartmentId: compartment\_ocid

clusterOptionId: "all"

title: "Kubernetes Version"

required: true

visible:

and:

- create\_new\_oke\_cluster

- show\_advanced

num\_pool\_workers:

type: integer

title: "Number of Worker Nodes"

minimum: 1

maximum: 1000

required: true

visible:

and:

- and:

- create\_new\_oke\_cluster

- not:

- cluster\_autoscaler\_enabled

The show\_advanced variable is of type boolean. Booleans are presented and rendered as check boxes that represent their value. Other variables can use the value of Boolean variables to determine their visibility and other properties. OCI resource variable types are special variables that provide querying capabilities into OCI resources.

The variable existent\_oke\_cluster\_compartment\_ocid is of type oci:identity:compartment:id. This is a hint for the Resource Manager to query the OCI compartment IDs and show them in a drop-down list when this variable is rendered on the UI. The user can select a compartment from the list of compartments shown, and the OCID for that compartment is then passed to the cluster.

Similarly, the existent\_oke\_cluster\_id is of type oci:container:cluster:id. This allows the user to choose a cluster by name from a list of existing Kubernetes clusters rendered as a drop-down list.

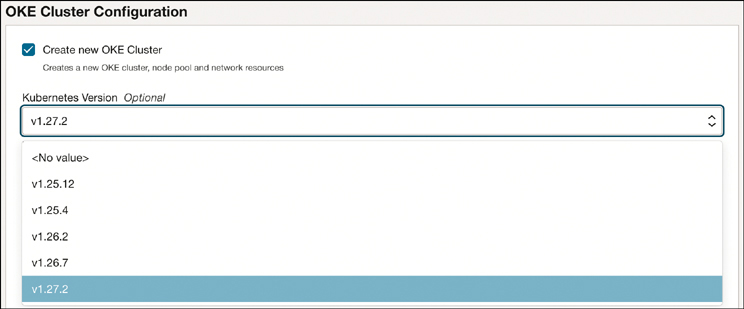
The variable k8s\_version is of a different type, called enum. Enums are enumerations of specific values, and this enum has four values. Resource Manager renders the enum as a drop-down list; the user can choose a value from this list that Resource Manager then provides to the Terraform variable. A selection from only these four values is possible, which avoids human errors and values that are beyond the expected set of possible valid values.

The num\_pool\_workers variable is of type integer, which means that it accepts only integer numbers. The UI that Resource Manager renders for these variables validates the input based on the type and its attributes. The integer type variable shown here has additional attributes, such as minimum and maximum.

This support for the notion of types in Resource Manager prevents the user from entering accidental typographical errors or incompatible values as if it were a free-form text field, as with the Terraform CLI. It also makes the process of entering values much easier for the end user. Values outside this range trigger a validation error on this component and inform users that they have made a choice outside the valid range.

Beyond the example in [Listing 2-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_17), several types exist, each one with various supported attributes. For a full description of these, you can look at the meta schema published by Oracle.[**6**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_6a)

[Figure 2-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02fig08) shows a portion of the UI rendered by the example in [Listing 2-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_17).



**Figure 2-8** UI Portion Rendered by the Configuration in [Listing 2-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_17)

The addition of metadata and types for variables enables Terraform developers to build stacks that are presented to end users with input fields that are less prone to human errors and typographical errors. Developers segregate input fields into groups, perform validation, and create more intuitive user interfaces that turn the Terraform configurations into deployment wizards that end users can use even if they have no prior experience with Terraform. Therefore, stacks with schema files allow solution developers to create polished and complete solutions that can be delivered directly from any web page using the Deploy button or through the OCI Marketplace.

**Publishing Your Stacks with Deploy Buttons**

The Resource Manager Service offers developers the capability to package Terraform-based solutions into stacks and publish these stacks over multiple channels for broad consumption. For commercial applications and licensed software, Oracle offers a partner program in which Oracle partners can list their stacks as solutions in the OCI Marketplace. Customers can consume these solutions by deploying them in their own tenancies directly from within OCI.

For open-source projects or other projects for which becoming an Oracle partner is not a consideration, the Resource Manager Service offers the capability to create web-based deployment buttons that can be placed on any web page. These deployment buttons are simple hyperlinks that can be placed on any web page; when clicked, they can load the stack into the Resource Manager Service and walk the user through the deployment process.

The Deploy button flow supports three providers:

* GitHub
* GitLab
* Oracle Object Storage’s preauthenticated requests

The developer builds a stack and creates a .zip file. The file is uploaded to any of the supported providers. The public URL for the ZIP file is used to construct the Deploy button. [Listing 2-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_18) shows the format for creating a Deploy button. The <package-url> is to be replaced with the public URL for the ZIP file from any of the supported providers. For GitLab and GitHub, Resource Manager supports both a direct link to the file or a link to a file published as part of a release. In most cases, it is preferable to use the link to a file published as part of a release in the Deploy button because that uniquely identifies a versioned artifact.

**Listing 2-18** A Stack Deploy Button in HTML

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f050-01a)

<a

href="https://cloud.oracle.com/resourcemanager/stacks/create?zipUrl=<package-

url>"

target="\_blank">

<img

src="https://oci-resourcemanager-plugin.plugins.oci.oraclecloud.com/latest/

deploy-to-oracle-cloud.svg"

alt="Deploy to Oracle Cloud"/>

</a>

Alternatively, you can use the markdown in [Listing 2-19](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_19) if you need to place a Deploy button for your stack in a markdown file, such as in your project README in GitHub.

**Listing 2-19** Stack Deploy Button in Markdown Format

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f050-02a)

[![Deploy to Oracle Cloud]

(https://oci-resourcemanager-plugin.plugins.oci.oraclecloud.com/latest/deploy-to-

oracle-cloud.svg)

]

(https://cloud.oracle.com/resourcemanager/stacks/create

?zipUrl=<package-url>)

When a Deploy button is clicked, the user is taken to the OCI console. If the user is not already logged in, the user logs into the tenancy and is immediately taken to the Create Stack workflow, where the stack is loaded from the provider and ready to be deployed. The user can then proceed with configuring the stack with the required variables and kickstart a Terraform apply job to create the resources and deploy the solution. An example of this is showcased in the example application for this book.

**Managing Multiregion and Multicloud Configurations**

All cloud vendors provide multiple regions, for better availability and fault tolerance. Managing these cloud infrastructure resources as code simplifies and automates their lifecycle management. With infrastructure defined as code and a full-featured CI/CD system managing the lifecycle, the next logical step is to manage your infrastructure across regions and across cloud providers using the same processes. With its provider-based architecture, Terraform enables you to easily create Terraform configurations that span multiple regions within the same cloud provider or even transcend cloud providers. This capability to define infrastructure in a parameterized way and create it in any region or cloud provider opens up new possibilities in implementing a globally distributed disaster recovery strategy. For instance, you can easily implement a pilot-light strategy with primary sites created from Terraform configurations operating at scale, while the same Terraform configuration with a lower number of resources acts as a pilot-light standby in another region, ready to be scaled up when required.

The providers used in a Terraform configuration are called out and declared within the configuration. The same Terraform configuration can have provider plug-ins for multiple cloud platforms. To create a Terraform configuration that spans multiple cloud providers, you can simply declare both provider plug-ins within the Terraform configuration. At runtime, Terraform will initialize all the declared providers. Resources from all the initialized providers can be used in the Terraform configuration.

Similarly, you also want to manage resources in two separate regions for the same cloud provider. Part of any provider configuration is the region for which the provider is configured. To work with multiple regions simultaneously, you can declare the cloud provider twice with *aliases*. Using aliases, you can determine which region you are interacting with for given resources. [Listing 2-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_20) presents an example.

**Listing 2-20** Multiple Provider Instances Can Be Instantiated with Aliases to Work with Multiple Regions or Tenancies

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f051-01a)

provider "oci" {

alias = "phoenix"

tenancy\_ocid = var.tenancy\_ocid

region = "us-phoenix-1"

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

provider "oci" {

alias = "ashburn"

tenancy\_ocid = var.tenancy\_ocid

region = "us-ashburn-1"

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

A resource block can include a provider argument, which uniquely identifies the provider to use for that specific resource. At runtime, when Terraform encounters a resource block with a provider argument, it maps to the provider definition with a matching alias and uses that provider instance to manage the resource lifecycle. In the example in [Listing 2-21](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_21), the resource block uses the argument provider set to oci.phoenix, which maps to the provider definition with the alias phoenix, which then uses the provider instance configured to use the Phoenix region.

**Listing 2-21** Resource Definitions Can Refer to Specific Provider Instances by Their Alias

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f052-01a)

resource "oci\_core\_virtual\_network" "my\_vcn" {

cidr\_block = var.vcn\_cidr

compartment\_id = var.compartment\_ocid

display\_name = var.vcn

dns\_label = "myvcn"

provider = oci.phoenix

}

As discussed earlier in the chapter, environment variables or the OCI CLI configuration file and its support for profiles can also be used in these cases to decouple the configuration values from the code itself. The OCI CLI configuration file shown in [Listing 2-22](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_22) describes two profiles names, IAD and PHX.

**Listing 2-22** OCI CLI Configuration File with Multiple Profiles

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f052-02a)

[IAD]

region=us-ashburn-1

tenancy=<tenancy\_ocid>

user=<user\_ocid>

fingerprint=<key\_fingerprint>

key\_file=<path to key file>

[PHX]

region=us-phoenix-1

tenancy=<tenancy\_ocid>

user=<user\_ocid>

fingerprint=<key\_fingerprint>

key\_file=<path to key file>

The Terraform provider configuration block shown in [Listing 2-23](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_23) refers to these profiles to load the provider configuration for Terraform. Using this approach, the actual values for authenticating with OCI can be maintained outside the Terraform code itself. The Terraform code can use the provider alias to distinguish between the two profiles. This makes the Terraform code more portable, by avoiding sensitive data from the codebase and making it possible to inject these values later. In the example in [Listing 2-23](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#list2_23), the two profiles represent two regions, and with this the Terraform code can work with OCI resources in both regions by referring to the appropriate provider alias.

**Listing 2-23** Provider Configuration with Multiple Regions

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02_images.xhtml#f053-01a)

provider "oci" {

alias = "phoenix"

config\_file\_profile= "PHX"

}

provider "oci" {

alias = "ashburn"

config\_file\_profile= "IAD"

}

**Summary**

This chapter examined how APIs are central to the cloud platform. Regardless of how the APIs are consumed, whether through the browser-based UI, command-line tools, or infrastructure-management tools such as Terraform, APIs form the primary control surface for interacting with the cloud platform. The chapter also served as a quick primer on Terraform, the infrastructure-management tool used throughout this book. You also briefly learned about the essentials of HCL, the language used for describing resource configurations in Terraform, and saw how Terraform integrates with multiple cloud providers using its provider-based model.

Additionally, the chapter covered the OCI provider for Terraform in detail, including ways to install, configure, and authenticate with OCI. You saw how the OCI resources are represented using the OCI provider and examined some strategies to decouple authentication information from the code for the infrastructure. You also investigated how Terraform tracks infrastructure state using state files and saw how to manage state files in OCI object storage when operating in teams.

Next, the chapter introduced the Resource Manager Service, a managed service for infrastructure management using Terraform. Resource Manager automatically tracks the Terraform state and makes team-based development of infrastructure easy and seamless. You also looked at additional features of Resource Manager, including drift detection, resource discovery, and generation of Terraform code as Resource Manager stacks from existing OCI workflows.

Furthermore, this chapter covered some of the supported providers inside Resource Manager, such as Kubernetes, and Helm, which later chapters discuss. You looked at the stack metadata, in which stacks with a schema file are presented as workflows with an intuitive UI to the end users. The stack metadata provides easy UI-based validations and lookup methods for several OCI resources, as well as common expressions. You also saw how developers can publish their Resource Manager stacks as Deploy buttons that can be placed on any web page. Finally, you looked at how you can work with multiple regions or multiple cloud providers.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_1) OCI Terraform provider: <https://registry.terraform.io/providers/oracle/oci/latest/docs>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_2) OCI Terraform provider GitHub: <https://github.com/oracle/terraform-provider-oci>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_3) OCI CLI installation and configuration: <https://docs.oracle.com/en-us/iaas/Content/API/SDKDocs/cliinstall.htm>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_4) Supported Terraform Providers in OCI Resource Manager: <https://docs.oracle.com/en-us/iaas/Content/ResourceManager/Concepts/providers.htm>

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_5) Extend Console Pages Using Schema Documents: <https://docs.oracle.com/en-us/iaas/Content/ResourceManager/Concepts/terraformconfigresourcemanager_topic-schema.htm>

[6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ref2_6) <https://docs.oracle.com/en-us/iaas/Content/ResourceManager/Concepts/terraformconfigresourcemanager_topic-schema.htm>