| **id** | **title** |
| --- | --- |
| module\_creation | Creating Terraform Modules |

**1. Creating Terraform Modules**

The Core Pipeline comes with a set of end-user modules to manage the lifecycle of a number of important cloud resources. These modules are intended to be used directly by applications onboarded to the Core Pipeline. In cases where the resource you want is not available as a consumable module, or you wish to compose other modules into larger units, you are encouraged to create your own. This section is intended to help you do just that.

**2. Audience**

Anyone who's interested in contributing to the set of Core Pipeline-supported Modules: platform engineers, LoB dev teams. Some knowledge of Terraform is very beneficial.

**3. Development Lifecycle Overview**

**3.1 Decide to use, modify or build:**

1. Identify the need for a new (or changed) module. Consult the Core Pipeline: [Terraform Module Catalogue](https://rbc-confluence.fg.rbc.com/display/AR/Core+Pipeline%3A+Terraform+Module+Catalogue) to see what modules are currently available, and what are in earlier lifecycle stages. If you don't see what you need there, check the module repo org for modules that may not yet be catalogued. If you don't find what you need, then:
2. Consult with the Product representative for the Core Pipeline to discuss your requirements.
3. If you still don't find what you need, then there would be a bigger conversation with the Core Pipeline team to dig into the technical requirements and come to a decision on how to proceed. This process would draw on resources such as the Azure Consumable Security Reviews to see if the resource in question is suitable for use from a security and risk perspective, but as a guide, this diagram illustrates the decision tree:

**3.2 Requirements identification and definition**

* What are the desired capabilities and features?
* Clearly identified use case
* Do we have something already available?
* How can we compose/exploit what we already have
* CCOs
* Design
  + Granularity and composition thoughts
  + Security
  + Associated policies

For an example of some of the above, see the [Azure SQL Managed Instance module](https://rbcgithub.fg.rbc.com/FRM0-Shared/terraform-azurerm-sql-managed-instance/blob/master/README.md)

**4. Design**

**4.1 Patterns**

Modules design must follow existing patterns and standards. See the sections on Source Repositories and Creation below for more information.

**4.2 Principles**

1. Modules should form low-level building blocks that can be composed by consuming organizations into larger modules.
2. Modules should be discrete with their dependencies externalized. If module depends on another resource, then it should not own that resource but should expect a reference (e.g., a name or ID) to be passed to it as a module input variable. For example, the Storage Account/Blob Storage module requires an Azure Keyvault to operate, but expects the Keyvault to have been created outside of the module (presumably by the Core Pipeline's Azure Keyvault module). This follows the dependency injection pattern, and supports principle 1. The reasons for this approach include:
   1. Simplicity: core pipeline modules should be only as complex as necessary
   2. Resource lifecycles are decoupled: bundling a resource such as an Azure Keyvault with a module such as the Storage Account means that the keyvault lifecycle is tied to that of the Storage account which is not necessarily a good thing. Architecturally, it makes more sense to allow the module consumer to define the Keyvault lifecycle separately from the modules that may use it.
   3. Core Pipeline modules are intended to be composed into larger units by consuming teams, such as customer engineering or a customer team. Bundling too many resources into a foundational module limits its reuse.

For some use cases that have a high degree of complexity, some of the above principles can be set aside. An example of this is a complex database server such as Azure SQL MI where there are data encryption keys, storage accounts, vulnerability assessments and various other resources that must be composed in a certain way to function correctly and meet the bank's security and operational requirements. In this case it may be acceptable to bundle more resources into one module.

**4.3 Source repositories**

Shared module code repositories are located in a cloud platform-specific Git organization. For Azure, this is [FRM0-Shared](https://rbcgithub.fg.rbc.com/FRM0-Shared), for AWS, [AWS0-Shared](https://rbcgithub.fg.rbc.com/AWS0-Shared), and for GCP, [GCP0-Shared](https://rbcgithub.fg.rbc.com/GCP0-Shared).

**4.4 Module Creation**

Once it has been decided that a new module is required, and the requirements and design have been completed the general approach to actually developing the code is:

1. Select a module to use as a template. Core Pipeline provides a [Git module template](https://rbcgithub.fg.rbc.com/FRM0-Shared/terraform-module-template) that provides examples of the structure, documentation and testing expected, but you can also choose a recent, real module as the basis for your new module. Some example include: [Azure SQL Managed Instance](https://rbcgithub.fg.rbc.com/FRM0-Shared/terraform-azurerm-sql-managed-instance) and the [Azure VM module](https://rbcgithub.fg.rbc.com/FRM0-Shared/terraform-azurerm-vm), but any recent, well documented module with comprehensive test cases may help.
2. Contact the Core Pipeline team and request that a new repository be created in one of the [cloud-platform]-Shared Git orgs. You may need to be added to the appropriate AD group to be able to contribute to this repo.
3. Use one of the above-mentioned sample repos and copy it to your new, empty repo (or if using the template one, select the "Use This Template" button)
4. Modify the code to match your requirements
5. Develop test cases and documentation
6. Publish and test
7. Get the various approvals from Security, SRE etc.

**5. Tagging and Versioning**

**5.1 When Publishing**

Modules are versioned based on [Terraform's Versioning Specification](https://www.terraform.io/plugin/sdkv2/best-practices/versioning). It is particularly important that versions are set to reflect any breaking changes so that consuming code that has not been properly constrained to use only a specific version (or only allow patch change) does not fail when it is run and picks up the latest, incompatible version of the module. For reference, important sections of the above document are reproduced here:

Observing that Terraform plugins are in many ways analogous to shared libraries in a programming language, we adopted a version numbering scheme that follows the guidelines of Semantic Versioning. In summary, this means that with a version number of the form MAJOR.MINOR.PATCH, the following meanings apply:

Increasing only the patch number suggests that the release includes only bug fixes, and is intended to be functionally equivalent.\*

Increasing the minor number suggests that new features have been added but that existing functionality remains broadly compatible.\*

Increasing the major number indicates that significant breaking changes have been made, and thus extra care or attention is required during an upgrade. To allow practitioners sufficient time and opportunity to upgrade to the latest version of the provider, we recommend releasing major versions no more than once per year. Releasing major versions more frequently could present a barrier to adoption due to the effort required to upgrade. Version numbers above 1.0.0 signify stronger compatibility guarantees, based on the rules above. Each increasing level can also contain changes of the lower level (e.g. MINOR can contain PATCH changes).

**5.1.1 Significant Version Numbers**

Version numbers below 1.0.0 are considered beta, and anything above that is considered GA (Generally Available).

**5.2 When Using**

When a module is used, best practices call for constraining the version of the module that the calling code will use. For example, this Terraform code constrains the version of the Azure VM module such that the major and minor versions must be 3 and 1 respectively, but the patch level can be anything greater or equal to 5. This is known as allowing the rightmost version component to increment and means that each time this Terraform code runs, it will pick up the latest patch of 3.1, but not 3.2, or 4.x.

module "vm" {

source = "localterraform.com/SHARED/vm/azurerm"

version = "~>3.1.5"

It is strongly recommended that your code either pins to a specific version of a module (e.g., version = "=3.1.5"), or uses the above constraint. If instead you simply accept the latest minor (or worse major) version, then you run the risk of changes to the consumed module breaking your code without warning.

For the full syntax, see [Terraform Version Constraints](https://www.terraform.io/language/expressions/version-constraints).

**6. Publishing and Deploying**

Modules are published to the Terraform Enterprise private registry by adding the appropriate version tags to a module repository release. Modules can only be published to the private registry in the SHARED TFE Organization. For example, in the UAT TFE instance, this is here: <https://uat.canadacentral.tfe.nonp.c1.rbc.com/app/SHARED/registry/private/modules>

**6.1 Initial Publishing**

The first step in publishing the initial version of a new module, is to add the module to Terraform's private registry. This involves navigating to the SHARED Org in TFE, then selecting Registry, then Modules, and finally `Publish':

The next step is to select the GitHub Organization containing the module's repository. TFE will already have been configured to access the FRM0/AWS0/GCP0-Shared orgs in the GitHub Enterprise selection, so select the appropriate source repo from the list presented:

Once the module has been published in this way, TFE will add a Git hook to the module s repository so that any new tagged releases are automatically published to TFE.

For example, to publish the 1.0.12 version of the terraform-azurerm-common module, go to its repo and under Releases, create a new release with a semantic version tag 1.0.12, on the master branch. As soon as the tag is created, the repo's Git hook will call TFE and the module will be published.

For more information, refer to [Terraform's documentation on module publishing](https://www.terraform.io/registry/modules/publish).

**7. Testing your module**

This section uses the Core Pipeline's VM Terraform module (terraform-azurerm-vm) as an example. When testing your own module, you will need to add your own module-specific tests, and a pipeline to run the test against the Core Pipeline's TFE environment.

**7.1 Testing Quick Start**

Follow these steps to get your module's test configuration up and running quickly. You still need to read the rest of this section, and understand how it works, but this will get you started.

**7.1.1 Assumptions:**

* The module's repo is already created (typically from the module template: FRM0-Shared/terraform-module-template)
* You have some familiarity with Terraform, Terratest and Golang

**7.1.2 Steps:**

1. Create (copy, or from scratch) a test/resources/main.tf. This is the TF code that uses the module to be tested. It is recommended that you use this code as an example when documenting your module, because it should exercise the features and capabilities of the module.
2. Create (copy, or from scratch) at least one Golang Terratest file. Each file typically contains a set of related tests and constitutes a test suite for the module. Examples include: terraform-azurerm-common/test/module\_test.go, and others in the test folders of each FRM0-Shared repo.
3. Write (or modify, if from a copy) the test cases. e.g., the lines starting at 47 in FRM0-Shared/terraform-azurerm-common/blob/master/test/module\_test.go
4. Create (copy, or from scratch) Jenkins file to run the test, and configure Jenkins to run the pipeline. Examples: FRM0-Shared/terraform-azurerm-common/test/Jenkinsfile, and terraform-azurerm-common
5. Run the job

Terraform modules are tested using the open source Terratest framework. This allows you to:

* Write some driver code that applies your shiny new module to a development environment in your already onboarded application space in Azure.
* Test if the resources created and configured by the module behave they way they are expected to
* For example, the tests for the terraform-azurerm-vm module do the following:
* Configure the actual module with parameters specific to the test environment
* Instruct Terraform to apply the module with the test configuration in the test environment
* Execute some smoke test cases against the new VM instance.

**7.2 Terratest concepts**

Terratest is a Golang library with test runners and helpers for a variety of cloud platforms and technology services. It's aimed at testing Terratest code, but has helpers to make it easier to integrate external tools and capabilities such as Docker, K8S, SSH, shell scripts etc.

Let's say you have written a Terraform module called start-web-app that composes an Azure resource group, VNET, Subnet, VM scaleset, NSG and an Application Gateway. You now want to do some unit and integration tests on the results of a Terraform Plan and Apply on the module. The general approach is to:

1. Write some test terraform code that uses your module
2. Write some Terratest Golang code that:
   * Runs Terraform plan and apply on the module, thus creating the resources in Azure
   * Runs a set of test cases against the results of the Terraform apply.
   * Tears down the system under test (the Azure resources created by the module)

In the Core Pipeline context, your modules and test code would be executed by the TFE environment, and you'd normally write a pipeline to run the test so it is (a) reusable, (b) can be run using the standard core pipeline context, and (c) can be triggered by Git actions and composed into larger test suites.

**7.3 Using Terratest in the Core Pipeline Context**

Testing Core Pipeline modules using Terratest presents some challenges around configuration and authentication. Terratest is mostly a wrapper around the Terraform CLI, and has no provision to use Terraform Enterprise's API and, right now, cannot be natively authenticated so it can use modules in the private registry – a problem because our modules under test depend on modules in the private registry.

These constraints also make it tricky to test a module by running Terratest locally on your dev system. Because of this, the recommended way to run tests is to run them in a Jenkins pipeline where some of these issues can be work-around. For example, access to the private registry can be achieved using TFE Auth (with the TFE admin token obtained from from the TFE data Keyvault. This is covered in the "Running your test" section below.

**7.3.1 Prerequisites**

* An onboarded dev app code and application
* Some Golang knowledge
* Some Terraform knowledge
* More Terratest Resources

Terratest has a comprehensive set of Azure helper functions that you can leverage from your Go code to simplify test creation.

For Azure modules, Microsoft has a guide to using Terratest here.

Another example of using Terratest on an Azure VM.

And, a very comprehensive guide to using Terratest in Azure in general, covering unit and integration tests.

**7.3.2 Unit vs Integration tests**

Terratest can perform 'unit' tests where you don't actually apply the Terraform modules, but just rely on the output of plan actions to validate module logic. While these can be important and useful, we tend to go straight to the integration test stage where we get Terraform to run an apply, and then test:

1. that the module created the expected resources correctly and then
2. that the resources behave as expected

**7.3.3 Test Structure in the Module Repo**

The (annotated and elided) file structure for a module's tests should be as follows

.

└── terraform-azurerm-vm

...

├── test # All tests and test fixtures/resources live under a test directory in the module's root directory

│ ├── Jenkinsfile # Pipeline to automate the test execution. Provides some of the variables such as the environment, app name.

│ ├── azurerm\_vm\_test.go # This is the main test driver, written in Golang and using the Terratest library.

│ │ # You can have as many test files as you need, depending on how comprehensive your test suite is

│ └── resources # The test fixtures live here. For Terratest, this consists mostly of Terraform code that uses

│ │ # the module under test, and simulates what a real consumer application would write.

│ ├── azurerm\_vm. # This is an Azure module, so we place our fixture code under a suitable-named directory.

│ │ ├── main.tf. # The Terraform code that will use the module under test

│ │ ├── terraform.tfrc.tpl # Template to configure the TF CLI (called by Terratest)

│ │ └── variables.tf # Optional. Additional variables related to the infrastructure surrounding the VM

...

**7.3.4 Configuring the test**

Most of the variables used in the test are set in the Jenkins test execution job, or in the TF code that uses the module under test:

1. test/resources/azurerm\_vm/main.tf. This is code that is very similar to that which a real consumer application would write
2. point the module to our actual module code under test

module "virtual\_machine\_linux" {

source = "./module"

deployment\_number = "1"

region = var.region

keyvault = var.keyvault

networking = var.networking

# Configure the VM kind, version etc.

soe\_distribution = "rhel\_7"

soe\_image\_version = "latest"

vm\_name = "soh-demo-vm"

vm\_size = "Standard\_D1\_v2"

vm\_admin\_username = "demouser"

linux\_vm\_public\_key\_secret\_name = "vm-public-key-soh1"

# Run a trivial HTTP server at boot time

# Don't do this on a production VM -- this is just so we can hit the VM with an HTTP request and see if it's running

# and reachable

boot\_command = "mkdir httpdir; cd httpdir; echo 'Hello, World!' > index.html; python -m SimpleHTTPServer 80"

}

output "vm" {

value = module.virtual\_machine\_linux.vm

}

For your module's test, adjust the various configurations to suite the module's variables, circumstances and your test environment.

**7.3.5 Running the Test**

Because our modules are intended to be applied in the context of the Core Pipeline and not locally (at least, not easily), the standard way of running tests is to use a Jenkins pipeline dedicated to testing this module. The pipeline will configure the module to run on a particular TFE instance, and will obtain all the necessary secrets required to run the module in the Core Pipeline-managed environment. The Jenkins file This example uses FRM0-Shared/terraform-azurerm-common/.../Jenkinsfile for now, because <https://rbcgithub.fg.rbc.com/FRM0-Shared/terraform-azurerm-vm/blob/master/test/Jenkinsfile> has not been converted to the new mechanism yet – TODO).

Let's look at the Jenkins file in some detail. This version has been annotated to help explain what is going on, and boilerplate sections have been elided. Some key points: Authentication with Terraform is done by setting the standard TF\_VAR\_tfe\_token to the value of the "tfe\_admin\_token" secret from the TFE instance's data keyvault. This grants access to the private registry. The resultant TFE config is written to a .tfrc file which is then used to automatically configure the CLI (which is invoked by Terratest) The entire RBC TFE bundle is downloaded from Artifactory and inflated into the build environment, so the build has access to the things it needs.

pipeline {

agent {

...

}

options {

withFolderProperties()

}

// Configure the runtime environment. Includes the Golang, Terraform Enterprise, the app identifiers

// and various Azure resource names and IDs

environment {

GOPATH = "${HOME}/go"

TFE\_BUNDLE = "1.0.17-rbc"

TF\_VAR\_TF\_LOG = "INFO"

TF\_VAR\_portfolio = "ccoe"

TF\_VAR\_service\_tier = "n"

TF\_VAR\_environment = "qat"

TF\_VAR\_tenant\_id = credentials('ARM\_TENANT\_ID')

TF\_VAR\_subscription\_id = "aec05452-daa7-4a8c-8b09-c9974ab436b3" // n0029

TF\_VAR\_client\_id = credentials('AEO0\_Platform\_Core\_ID')

TF\_VAR\_client\_secret = credentials('AEO0\_Platform\_Core\_Secret')

TF\_VAR\_tfe\_hostname = "qat.canadacentral.tfe.nonp.c1.rbc.com"

TF\_VAR\_onboard\_hostname = "core-pipeline-onboard-frm0-private-qat.canadacentral.aks.nonp.c1.rbc.com"

TF\_VAR\_app\_code = "${getAppMetadata()[0]}"

TF\_VAR\_app\_name = "${getAppMetadata()[1]}"

TF\_VAR\_branch = "${getAppMetadata()[2]}"

target\_operating\_system = "linux"

target\_platform = "amd64"

core\_pipeline\_provider\_version = "1.0.0"

core\_pipeline\_provider\_path = "~/.terraform.d/plugins/rbc.com/api/corepipeline/${core\_pipeline\_provider\_version}/${target\_operating\_system}\_${target\_platform}"

TFE\_SUBSCRIPTION\_ID = credentials('SHARED\_SERVICES\_SUBSCRIPTION\_ID')

TFE\_DATA\_RESOURCE\_GROUP\_NAME = "${TF\_VAR\_service\_tier}-nonp-ccoe-frm0-tfe-${TF\_VAR\_environment}-data-rg-cac-1"

}

stages {

// This stage builds custom plugins and makes them available for Terraform locally - yes, it is required by remote backend for some reason.

stage("Build Core Pipeline Provider") {

...

}

// This stage copies all module's files and directories into the test folder

// in order to make Terraform upload the module's source code to TFE alongside the test code.

// Grab the entire RBC Terraform bundle from Artifactory, and inflate it locally.

// This makes all these modules etc. available to TF code being run by Terratest

stage("Copy Module(s)") {

steps {

sh """

curl -k -O https://rbcartifactory.fg.rbc.com/artifactory/generic-aeo0/terraform/terraform-version-${env.TFE\_BUNDLE}.zip && unzip terraform-version-${env.TFE\_BUNDLE}.zip

rm -rf \*.tar.gz

cp -r ./plugins ~/.terraform.d/ && rm -rf ./plugins

ls -l ~/.terraform.d/plugins

mkdir -p /tmp/.terraform.d/plugin-cache

terraform --version

"""

}

}

stage("Resolve Test Dependencies") {

steps {

dir("test") {

sh "go get -t -d"

}

}

}

stage("Configure Terraform") {

// Get the TFE config we need, and save it to a TF CLI config file (.tfrc) where it will be available

// to the TF CLI when it is invoked by Terratest

steps {

dir("test/resources/main") {

script {

azLogin()

tfe\_keyvault\_name = getTfeKeyvaultName()

terraformrc = getKeyvaultSecret("terraformrc", tfe\_keyvault\_name)

tfe\_admin\_token = getKeyvaultSecret("tfe-admin-token", tfe\_keyvault\_name)

def binding = [

"tfe\_hostname": env.TF\_VAR\_tfe\_hostname,

"tfe\_token": "${tfe\_admin\_token}"

]

def template = readFile("./terraform.tfrc.tpl")

def text = renderTemplate(template, binding)

writeFile(file: "./terraform.tfrc", text: text)

}

}

}

}

stage("Test") {

// Now we can run Terratest, using the various configurations set earlier on.

//

steps {

dir("test") {

script {

azLogin()

tfe\_keyvault\_name = getTfeKeyvaultName()

onboard\_auth\_id = getKeyvaultSecret("onboard-auth-id", tfe\_keyvault\_name)

tfe\_admin\_token = getKeyvaultSecret("tfe-admin-token", tfe\_keyvault\_name)

withEnv([

"TF\_VAR\_onboard\_auth\_id=${onboard\_auth\_id}",

"TF\_VAR\_tfe\_token=${tfe\_admin\_token}",

"PORTFOLIO=${env.TF\_VAR\_portfolio}",

"APP\_CODE=${env.TF\_VAR\_app\_code}",

"APP\_NAME=${env.TF\_VAR\_app\_name}",

"BRANCH=${env.TF\_VAR\_branch}",

"SERVICE\_TIER=${env.TF\_VAR\_service\_tier}",

"ENVIRONMENT=${env.TF\_VAR\_environment}",

"API\_HOSTNAME=${env.TF\_VAR\_onboard\_hostname}",

"ARM\_TENANT\_ID=${env.TF\_VAR\_tenant\_id}",

"ARM\_SUBSCRIPTION\_ID=${env.TF\_VAR\_subscription\_id}",

"ARM\_CLIENT\_ID=${env.TF\_VAR\_client\_id}",

"ARM\_CLIENT\_SECRET=${env.TF\_VAR\_client\_secret}",

"ONBOARD\_AUTH\_ID=${onboard\_auth\_id}",

"TF\_LOG=${env.TF\_VAR\_TF\_LOG}"

]) {

sh "curl https://${env.TF\_VAR\_onboard\_hostname}/health && go test -count=1 -timeout 30m -p 1"

}

}

}

}

}

}...

The meat of the test is the Golang Terratest code, located in test/azurerm\_vm\_test.go. The version shown below has been annotated to help illustrate what's going on.

...

func TestAzurermVm(t \*testing.T) {

...

// Point Terratest at the workspace fixture

workspaceOptions := terraform.WithDefaultRetryableErrors(t, &terraform.Options{

TerraformDir: "./resources/workspace",

NoColor: true,

})

// Tell Terraform to destroy the deployment no-matter what the outcome of the test

defer terraform.Destroy(t, workspaceOptions)

// Tell Terraform to run an 'init' and then an 'apply' on the workspace fixture code.

terraform.InitAndApply(t, workspaceOptions)

// Configure options for the actual VM module

azurermVmOptions := terraform.WithDefaultRetryableErrors(t, &terraform.Options{

TerraformDir: "./resources/azurerm\_vm",

BackendConfig: map[string]interface{}{"token": os.Getenv("TF\_VAR\_tfe\_token")},

NoColor: true,

})

// And a similar set of actions on the VM module

defer terraform.Destroy(t, azurermVmOptions)

// This will block here and may take some time, because this is where Terraform

// is inflating the resources in your module

terraform.InitAndApply(t, azurermVmOptions)

vm := make(map[string]interface{})

// Terraform has finished applying the module. Let's grab the module output so we can sanity test what it has done

terraform.OutputStruct(t, azurermVmOptions, "vm", &vm)

vmId := vm["id"].(string)

vmPrivateIpAddress := vm["private\_ip\_address"].(string)

// Run some actual tests. In this case we're just ensuring that the VM was created, by testing if the

// outputs are valid.

assert.NotEmpty(t, vmId)

assert.NotEmpty(t, vmPrivateIpAddress)

expectedVmId := fmt.Sprintf(

"/subscriptions/%s/resourceGroups/%s-%s-ccoe-frm0-terraform-azurerm-vm-%s-rg-cac-1/providers/Microsoft.Compute/virtualMachines/soh-demo-vm",

os.Getenv("TF\_VAR\_subscription\_id"),

os.Getenv("TF\_VAR\_service\_tier"),

os.Getenv("TF\_VAR\_environment"),

os.Getenv("TF\_VAR\_branch"),

)

// and that the VM ID matches what we expect

assert.Equal(t, vmId, expectedVmId)

// So far, we've just checked that Terraform did its job correctly

// Now let's run an actual smoke/integration test against the running VM.

// Here we use Terratest's http\_helper to issue an HTTP request against the private IP of the

// VM (obtained from the module's outputs above), and see that it returns what we expect.

// If you look at the main.tf code that created the VM, you will see that it ran a boot script

// that started a trivial Python HTTP server that responds with "Hello, World"

// We can get quite fancy here and use Terratest to exercise all manner of features and capabilities

// of our module.

// http\_helper will fail the test if it doesn't get the expected HTTP 200 code, so no need

// to have an assertion here.

// See https://github.com/gruntwork-io/terratest/blob/master/modules/http-helper/http\_helper.go#L100

http\_helper.HttpGetWithRetry(t, fmt.Sprintf("http://%s", vmPrivateIpAddress), nil, 200, "Hello, World!", 10, 3\*time.Second)

}

// Note that we're not tearing own the resources here, because Terratest will do that when the test code exits.

**7.4 More complex tests**

Terratest has a number of capabilities that are very useful in more complex test scenarios. For example, the code above will automatically tear down the deployment when it exists (because of the 'defer terraform.Destroy(t, azurermVmOptions)' lines), but in cases where your module creates a more complex set of resources and those resources take a lot of time to inflate, then you can ask it not to run a destroy action on exit by using test stages. This allows you to run one test suite or case to inflate the environment, then another set of test cases or a suite repeatedly over the same environment. See Iterating locally using test stages for more information.

Similarly, Terratest has a number of helpers that can simplify running functional tests gagainst the inflated resources. The example above shows the use of the http\_helper to hit the VM's test endpoint and verify that it is reachable and behaves correctly, and you can do similar things with the ssh\_helper if you want to test ssh connectivity.