

How to Test Terraform Code

DevOps World is Full of Fear



The Plan

- ◆ Manual tests
 - Manual testing basics
 - Cleaning up after tests
- ◆ Automated tests
 - Unit tests
 - Integration tests
 - End-to-end tests
 - Other testing approaches

Manual Tests

- ◆ What does manual testing mean in Terraform?
 - How much is it like manual testing in a programming language?
- ◆ With terraform, there is no quick and dirty test environment like *localhost* for web apps
 - True of most IaC tools
 - Deployment needs to be a real environment
 - Testing should *never* be done in a production environment
- ◆ It is essential to have examples which can be manually tested in a real AWS environment

Test Frameworks

- ◆ A test harness or driver automates the running of the tests
 - Involves some appropriate client to validate the result of the test
- ◆ For web apps, 'curl' or some other tool can be used to check the output at a specific address or URL
- ◆ If possible, the checking of the results should be automated and not require manual inspections
- ◆ Best practice is to set up a test sandbox
 - There will be a lot of building and tearing down of code
 - Each developer should have their own sandbox
 - The Gold standard would be separate AWS accounts

Cleaning Up After Testing

- ◆ Regularly clean up the testing sandbox environments
 - Running deployments cost money
 - It's easy to overlook infrastructure so that it sort of just hangs around
- ◆ At a minimum, use *terraform destroy* after completing the testing
 - Also consider a regular "scrubbing" of the workspace using a cron job
- ◆ Some useful tools to do an account resource "purge"
 - **cloud-nuke** : An open source tool that can delete all the resources in your cloud environment
 - **Janitor Monkey** : An open source tool that cleans up AWS resources on a configurable schedule
 - **aws-nuke** : An open source tool dedicated to deleting everything in an AWS account

Automated Testing

- ◆ There are three kinds of automated tests:
- ◆ Unit tests
 - Unit tests verify the functionality of a single, small unit of code
 - External dependencies are replaced with test mocks, also called "stubs"
- ◆ Integration tests
 - Integration tests verify that multiple units work together correctly
 - Other parts of the system not being tested are mocked out
- ◆ End to end testing
 - End-to-end tests involve exercising the entire architecture from the end-user's perspective
 - E2E testing runs with no mocks in an architecture that mirrors production

Unit Testing

- ◆ The first step is to identify what a terraform “unit” is
 - A unit would be a single generic module like those developed in a previous module
 - The idea of unit testing traditionally assumes a procedural programming language
 - The idea of unit testing has to be tweaked to work with declarative languages
- ◆ The unit under test is being deployed into a real AWS environment
 - Unit tests in terraform have to involve some integration with AWS resources
 - Carefully controlling the AWS environment set up can be thought of as a way to *mock*

Unit Testing Strategy

- ◆ A basic strategy for writing unit tests for Terraform is:
 - Create a generic, standalone module
 - Create an easy-to-deploy example for that module
 - Run *terraform apply* to deploy the example into a real environment
 - Validate that what you just deployed works as expected
 - Run *terraform destroy* at the end of the test to clean up.
- ◆ For automated testing, first write the manual test, then automate
 - Remember to try the automated tests on code that has no errors to ensure the tests are being automated correctly

Rethinking Unit Testing in Terraform

- ◆ Unit testing has traditionally been functional testing
 - The correctness of the code is inferred from:
 - Specifying expected outputs for each test case
 - Running the unit under test with the test case inputs
 - Recording the actual outputs
 - Comparing actual outputs against expected outputs
- ◆ Exactly the same approach can be used in terraform
- ◆ Given a terraform module, a description of what is expected can be written
- ◆ Using *outputs* , the actual result can be documented
- ◆ The difference between the actual and expected can be computed
- ◆ For terraform, unit tests should verify:
 - Was the right resource created?
 - Does it have the correct expected attributes?
 - Are the right associations in place

Automated Testing Pointers

- ◆ Manual testing should be done in a sandbox account
 - For automated testing, this is even more important
 - A totally separate account is recommended.
- ◆ As an automated test suite grows, hundreds or thousands of resources may be created in every test suite, so keeping them isolated from everything else is essential
 - Teams should consider a completely separate environment just for automated testing
 - This is separate even from the sandbox environments you use for manual testing.

Testing Accuracy

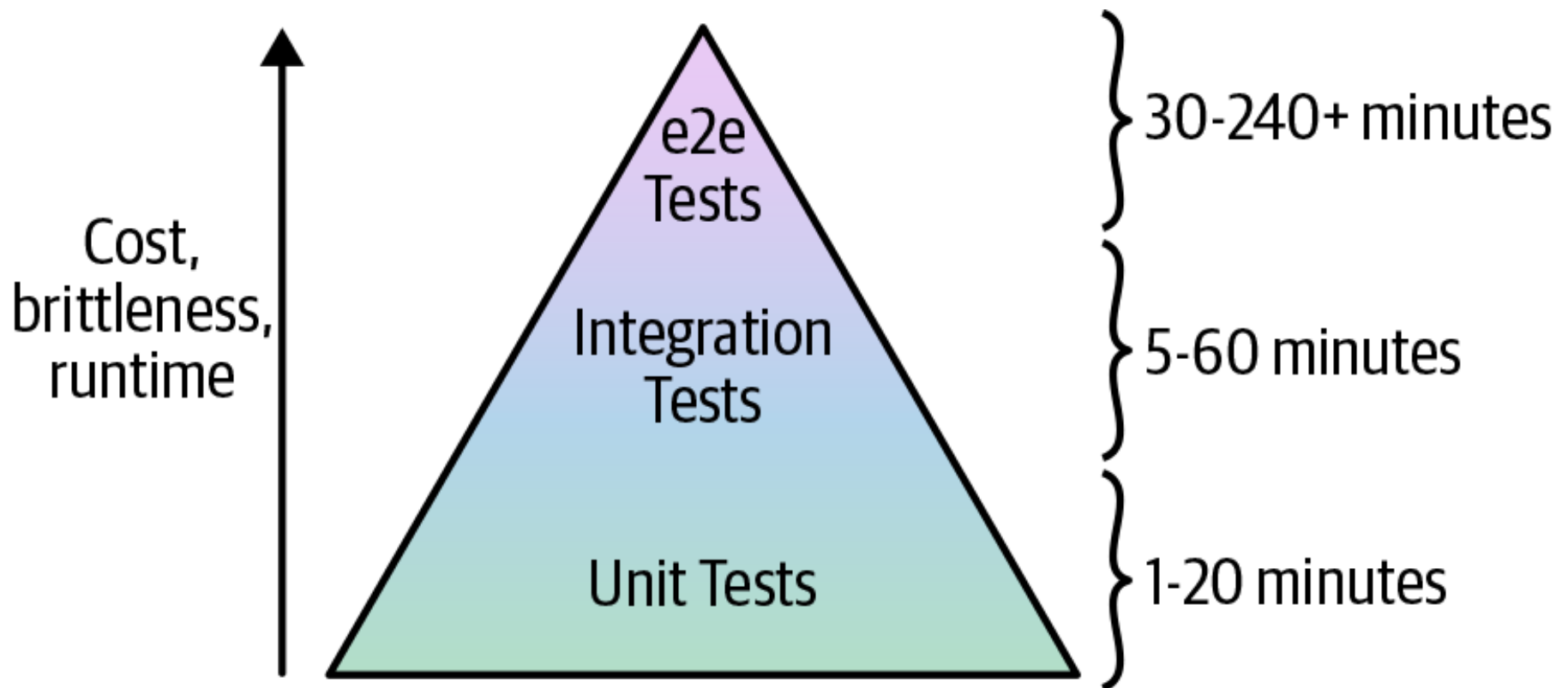
- ◆ When testing, two types of testing errors can occur
 - *False Positives* : The test fails but it should have passed because there is no underlying fault
 - *False Negatives* : The test passes but it should have failed because it didn't detect an underlying fault
- ◆ When automatic tests, errors can occur in the code being tested or the test automation code
- ◆ Before running any automated tests
 - Have a terraform benchmark configuration with no errors
 - Have a set of error-benchmark modules with deliberate errors, if possible
 - Run any test automation scripts against the benchmark to identify possible false positives
 - Run the scripts against specific error modules to identify possible false negative

High Quality Testing

- ◆ Testing should never be ad hoc
- ◆ A third source of testing errors are poor-quality tests
- ◆ Test need to satisfy certain criteria:
 - *Validity* : The tests actual test what we *think* they test
 - *Accuracty* : Discussed on the previous slide
 - *Reliability* : Results of the tests are not affected by extraneous factor, like the AWS account the tests are being run in
 - *Comprehensive* : Nothing can "fall through the cracks" during testing
 - *Economical* : The set of tests is minimal without compromising any of the above
- ◆ A software tester or QA team member should be part of any testing effort

End to End Tests

- ◆ As our tests include more and more code, they become longer to execute and more costly to set up and run
- ◆ We should plan for a large number of unit tests, smaller number of integration tests and even smaller number of end-to-end tests



End to End Setup

- ◆ With larger and more complicated infrastructure, setting up a stable infrastructure (test environments, namespaces et) becomes increasingly difficult
 - Do as much of your testing as low in the pyramid as possible
 - The bottom of the pyramid offers the fastest, most reliable feedback loop
- ◆ Deploying a complicated architecture from scratch is untenable for several reasons
- ◆ Too slow:
 - The more complex the infrastructure, longer it takes to set up
 - Limits the amount of testing that can be done which means slow feedback
- ◆ Too brittle:
 - Constantly redeploying a complex setup increases the likelihood of transient errors
 - This means constant retries which inhibit the whole testing effort

End to End Strategy

- ◆ A common end-to-end strategy is:
 - A persistent, production-like environment called “test” is deployed which is left running
 - Every time a change is made to the infrastructure, the end-to-end test does the following:
 - Applies the infrastructure change to the test environment
 - Runs validations against the test environment (e.g., use Selenium to test your code from the end-user’s perspective) to make sure everything is working
- ◆ More closely mimics how changes will be deployed in production
 - Also confirms the deployment process also works - for example, the change can be made with zero downtime

End to End Testing

- ◆ E2E testing treats the AWS environment as a black box
- ◆ Scenarios are selected and developed that:
 - Represent all the functionality required from the client perspective
 - Are executable through the application interfaces
 - This allows regression testing after changes to the internal structure of the AWS environment
- ◆ Testing can be done with standard E2E tools like Functional Tester and Selenium

Static Analysis

- ◆ Static analysis involves running tools that examine the structure of the code without executing it
- ◆ Common tools are:
 - *terraform validate* : a command built into Terraform that you can use to check your Terraform syntax and types
 - *tflint* : A “lint” tool for Terraform that can scan Terraform code and catch common errors and potential bugs based on a set of built-in rules
 - *HashiCorp Sentinel* : A “policy as code” framework that allows you to enforce rules across various HashiCorp tools

Property Testing

- ◆ These are testing tools like `rspec-terraform` that use Domain Specific Languages to confirm that the infrastructure conforms to a specification
- ◆ For example:

```
1 describe file('/etc/myapp.conf') do
2     it { should exist }
3     its('mode') { should cmp 0644 }
4 end
5
6 describe apache_conf do
7     its('Listen') { should cmp 8080 }
8 end
9
10 describe port(8080) do
11     it { should be_listening }
12 end
```

Key Takeaways

- ◆ When testing Terraform code, testing is done in a real environment
 - All manual testing is done by deploying real resources into one or more isolated sandbox environments
- ◆ Sandbox environments *must* be purged after testing
 - Otherwise, the environments will become unmanageable, and costs will spiral out of control
- ◆ Unit testing for terraform involves automatically or manually checking deployed configurations for correctness
 - All automated testing is done by writing code that deploys real resources into one or more isolated sandbox environments
- ◆ Smaller modules are easier and faster to test
 - Smaller modules are easier to create, maintain, use, and test.
- ◆ End-to-end testing is done using a long-lasting near production environment

Lab

- ◆ Please do Lab 7-1