Learn Terraform

In my last post, Learning Terraform, I committed to learning Terraform. I've started reading Terraform: Up and Running: Writing Infrastructure as Code. It has given me a great primer thus far. I'm converting my Cloudformation templates to Terraform. I'm all in at this point and am convinced that this is my preferred tool for building infrastructure as code.

The Basics

You'll want to install the Terraform command-line interface. I ran into some issues with the Homebrew version of Terraform, so I installed the binary. Terraform provides a single binary with zero dependencies, which warms my heart.

I love shell aliases. So, I set up the following aliases for my terraform activities

```
alias tf='terraform'
alias tfa='terraform apply'
alias tfd='export TF_LOG=DEBUG'
alias tfdl='unset TF_LOG'
alias tfdr='terraform destroy'
alias tfp='terraform plan'
alias tfs='terraform show'
alias tft='export TF_LOG=TRACE'
```

I've found these to be the most common commands I'm running while developing new templates.

If you're on Windows 10 you can install a linux shell or if you're stuck in the past you can use Cygwin.

On either platform add the aliases to your .bash_profile then run this command.

```
source ~/.bash_profile
```

I recommend using a Terraform IDE extension for completion and syntax highlighting.

- Jetbrains Products
  - HashiCorp's IntelliJ plugin
- Visual Studio Code
  - Terraform editing features
  - Terraform doc snippets

Don't forget the golden rule of Terraform before diving too deep.

The master branch of the live repository should be a 1:1 representation of what's deployed in production.

If you're chanting incantations to get your Terraform managed resources deployed, step back and rethink your approach. Following this rule will also allow Terraform to serve as documentation.

Testing

Like any other code starting with tests is a great idea. Proving your code is doing what it's supposed to do is a cornerstone of great software.

Testing Terraform requires the following.

- Multiple Terraform state environments along with a test and production environment.
- Input variables that allow you to change environment-specific values such as connection strings and keys

The tests you write should verify that your resources made it to the environment you are targeting. This can be done by making use of output values fed to simple scripts that check if resources exist after deployment.

Tools like Terratest can give you a leg up if you want to get more sophisticated.

Providers

Learning Terraform requires an understanding of providers. They are a key feature that harnesses the true power of Terraform. The collection of providers that Terraform supports are numerous. I've tried out the New Relic and AWS providers with great success.

Importing a New Relic dashboard went surprisingly smoothly. It was a bit painful picking apart the extra attributes after import but overall, it saved time. I've added a new Terraform feature proposal to GitHub. Upvote that issue if you're interested in saving even more time.

Data Sources

Data sources are a key concept that allows you to query resources using various APIs. These read-only resources are usually created outside of your Terraform module. They may have been created manually or in another terraform module.

Terraform data sources are necessary for resources that are not managed by Terraform.

The syntax is super simple. This code example, from the Terraform website, is the perfect use case. If we would like to query an AMI to associate with a launch configuration this is how we do it.

```
# Find the latest available AMI that is tagged with Component
= web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Stack Overflow has a post that answers the Terraform data source use cases in more detail.

State

Terraform's magic is its ability to manage resource state. State management can be painful. Kudos to Hashicorp for taking on the state management challenge. If you're working on a team this feature is necessary to ensure the stability of your resources. If you're not working on a team you can get by without considering this feature.

If we're a team managing state, we'll need to make sure we don't cross the streams. Luckily Terraform provides the ability to store your state in a remote data source. This allows you to edit the same resources on a team by storing the state in a central location.

There are multiple backends that can store Terraform's state.

The most common backend is S3. It's simple and works, and this is how it's done.

```
terraform {
  backend "s3" {
    bucket="<Bucket Name>"
    key="<Bucket Key Where State Will Be Stored>"
    region="us-east-1"
  }
}
```

The only thing you need to do is add that to your Terraform template and you're done. If you add this later, you'll need to reinitialize the terraform state.

Remember

- Some Terraform state is eventually consistent. If a resource fails to deploy, you'll need to run it again after fixing the problem. Use the depends_on meta-argument to avoid some of this back and forth work.
- Valid plans can and will fail. Terraform can't handle every edge case in the universe. Failures are usually caused by not importing existing resources.

- Commit to **only** using Terraform to manage your resources. If you use user interfaces instead weird errors will occur while using it.

Modules

A Terraform module is a directory after running this command.

```
terraform init
```

We're all using Terraform modules by way of using the Terraform command-line interface. Terraform input variables and output values control your module's behaviors. Access child module outputs with this syntax.

```
module.MODULENAME.OUTPUTVARIABLENAME
```

The calling module should handle the provider definition. Here's an example of calling a module, we already know what a module looks like.

```
provider "aws" {
    region = "us-east-1"
}

module "webserver_cluster" {
    source = "../modules/services/webserver-cluster"
}
```

This code assumes your modules folder lives outside of the Terraform template you are working on.

That’s about all there is to using a module although inputs and outputs will again come into play.

Remember

- When creating a reusable module, always prefer using a separate resource. Separating resources will allow callers of your module to extend your module with custom rules.
- Version your modules to avoid breaking dependent code
- Make your modules configurable for added flexibility

Import

Behold the Terraform import command.

```
terraform import aws_s3_bucket.jeffbaileywebsite jeffbaileywebsite
```

Run this command when you have existing resources you would like to manage with Terraform. If you’re migrating from CloudFormation templates, this command will be your best friend.

Importing is awkward, but adding the resource with a local

name will allow you to run the import command.

```
resource "aws_s3_bucket" "bucket" {
}
```

Once complete, you can run this command to get a representation of the resource you are importing.

```
terraform plan
```

If you set up the aliases above, you can simply type tfs to run the same command.

Copy the output of the resource you are importing and run tfs again. This will give you complaints about fields like **id** that can't be set in a Terraform template. Remove the invalid fields and run tfs again to see if your local state is valid. If it is, you can run this command.

```
terraform apply
```

Check that you haven't deleted your infrastructure then commit your template. Now you're off to the races with Terraforming further changes to your resources in the future.

Challenges

Any tool comes with its challenges. Here are some of the

problems you will encounter.

Problem	Solutions
Avoiding an override of your remote state with local state	Deploy changes with a build pipeline that only allows one deployment at a time Use remote state and diligently run the terraform plan command locally to capture the latest changes
Ensuring resources are in the state they are supposed to be in	Create unit and integration tests to validate that your resources deployed as expected Use an isolated testing environment to validate all your changes and run your tests

Small challenges for great rewards

Conclusion

While learning Terraform might save your life it’s not all roses and sunshine. There will be problems using it like any other tool.

While creating an AWS Cost and Usage Report an internal server error occurred. The aws_cur_report_definition failed to deploy unless targeting the us-east-1 region. When I changed the template to target us-east-2 instead of us-west-2, everything worked. CloudFormation might have been more helpful.

Adoption of Terraform within your team will need a culture change. The team will need to understand and appreciate the

benefits of Terraform. Editing a dashboard in a slick user interface is convenient. While it's convenient to edit a dashboard, it doesn't share the intent with other team members. Pull requests will prompt your team to question a dashboard change. Your team will also have an opportunity to learn about new features added to a dashboard.

The bottom line

Changing a dashboard can cause your employer to lose millions in lost revenue. If your widget says everything is fine, but it's not, was the convenience of the UI worth the cost? Make sure your team sees the value before asking them to delve into Terraform.

Overall Terraform is great and is getting better. I'm committed to using it for my IaC efforts going forward.

Continue Learning Terraform

If you work for a company that has stringent compliance workflows watch this video from Ellie Mae. These guys automated pretty much everything to capture every change everywhere.