Using Variable-Driven AWS Configuration with Terraform Enterprise and Github

Infrastructure-as-Code is an excellent way to better represent and manage infrastructure. AWS is particularly easy to provision and manage programmatically using Terraform Enterprise which is a collaborative, cloud-hosted version of the popular Terraform OSS product. Another big advantage with the Enterprise or Cloud products is the ability to use externally facing APIs and integrations to make your codified infrastructure more dynamic.

I'm working on a specific use-case where I have some AWS components that I also have an external service (in this case, Turbonomic) which is going to define the sizing and scaling for. Another blog will come shortly on that interaction but for this example I just want to show you my core configuration between AWS, Terraform Enterprise, and Github.

Here is the flow that we want to achieve:

1. Describe our AWS infrastructure using variable-driven configuration in Terraform Enterprise (TFE)
2. Update the configuration variables using the Terraform API
3. Send an update to the Github repository via the Github API to trigger a webhook to TFE

Ingredients for our Recipe

You're going to need a few things to start, including a Terraform Enterprise account. The functions you will tap into here also work on Terraform Cloud as well although I'll share specifics on that flow in a different blog

Next up in our ingredients list is:

1. Github API key for OAuth access and the ability to write to a repository
2. AWS client and secret key with the necessary IAM privileges to manage resources you will deploy
3. Shell access on a workstation or server to run Bash scripts – both of which are included in the Github repo

The sample Github repo I'm using is here: https://github.com/discoposse/tfe-turbonomic-demo

There's a demo video at the bottom which shows the whole process as it happens so you can see the flow.

Setting up your Terraform Configuration

It's always best to start simple. The basic configuration I'm showing you here is an EC2 instance which is running inside a VPC that I want to be sized using an Input Variable rather than codifying the instance type in the Terraform configuration statically.

There are three files in use in my Terraform configuration which is hosted on Github (https://github.com/discoposse/tfe-turbonomic-demo) which are:

1. **provider.tf** – defines the AWS provider and the backend configuration

2. **vars.tf** – variable definition for what will be passed to Terraform during provisioning
3. **main.tf** – code to define the AWS infrastructure which happens to be an EC2 instance

Provider info is easy to understand as it's just assigning the access and secret key plus the region, all three of which are pulling from input variables:

```
provider "aws" {
  access_key = "${var.aws_access_key}"
  secret_key = "${var.aws_secret_key}"
  region = "${var.aws_region}"
}
```

The backend configuration to associate this with Terraform Enterprise is done using this stanza:

```
terraform {
  backend. "remote" {
  organization = “Turbonomic"
  workspaces {
    name = "tfe-turbonomic-demo"
    }
  }
}
```

This contains specific organization and workspace assignments because the version control system (VCS) will be Github and this code is only assigned to one workspace.

Variables are defined using vars.tf including mandatory configuration parameters for AWS to deploy our EC2 instance. The one variable of note is the `turbonomic_instance_size` which is referred to in the `main.tf` file.

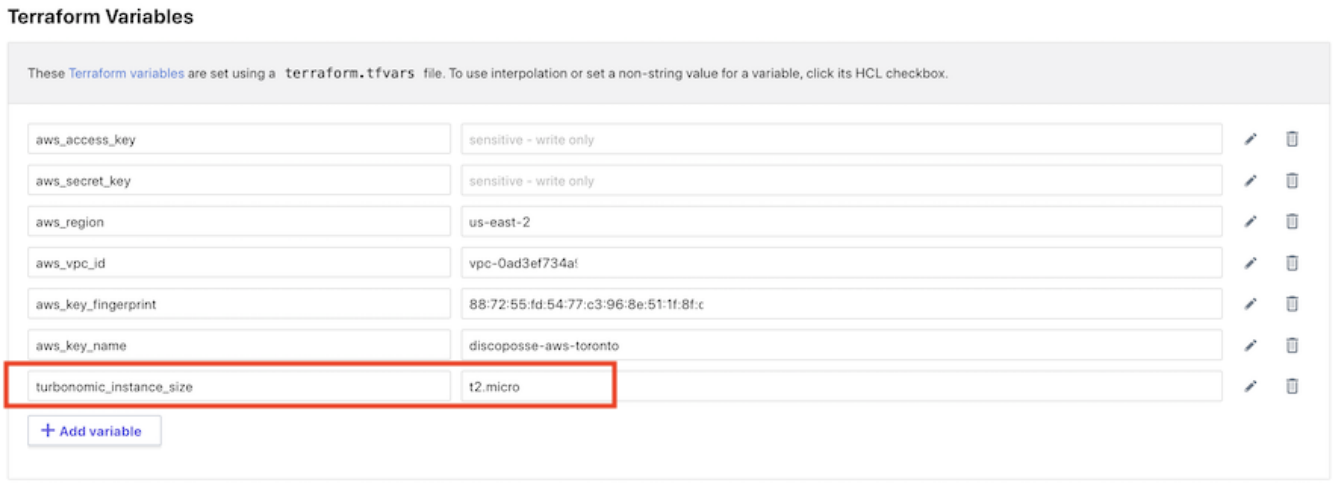
```
# main.tf
resource "aws_instance" "web" {
  ami = "ami-c55673a0"
  instance_type = "${var.turbonomic_instance_size}"
  key_name = "${var.aws_key_name}"

tags = {
  Terraform = "true"
  ProvisionedBy = "Project Terra"
  Turbonomic = "true"
  }
}
```

You can see we are creating an EC2 instance from an AMI (which can be coded like the example here or assigned dynamically as well) and the instance type and size will be pulled from the `turbonomic_instance_size` input variable during each plan/apply.

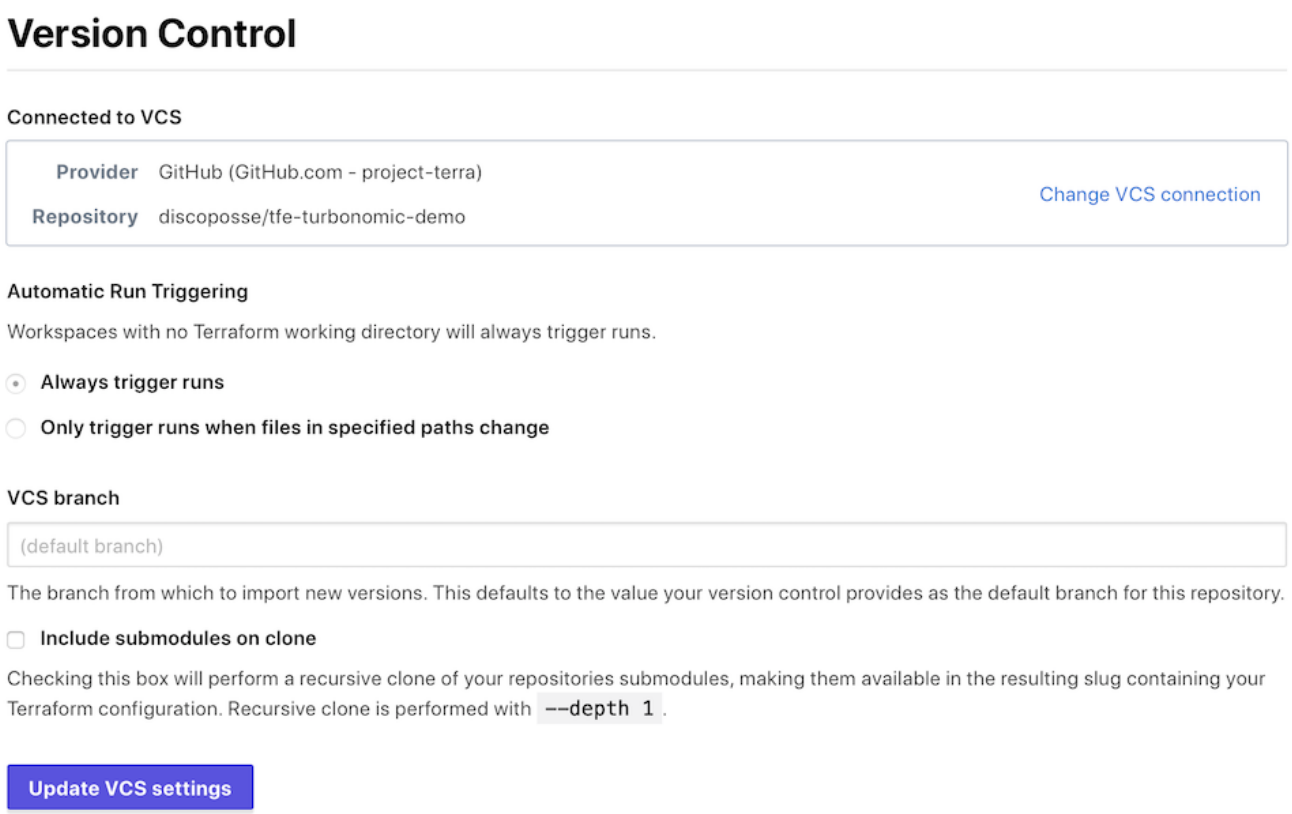
Setting up the Input Variables in Terraform Enterprise

TFE allows you to share configurations and state so this is where we centralize things rather than having locally managed environment variables or `.tfvars` files. You need to create a variable in the workspace for each defined variable in the configuration files.



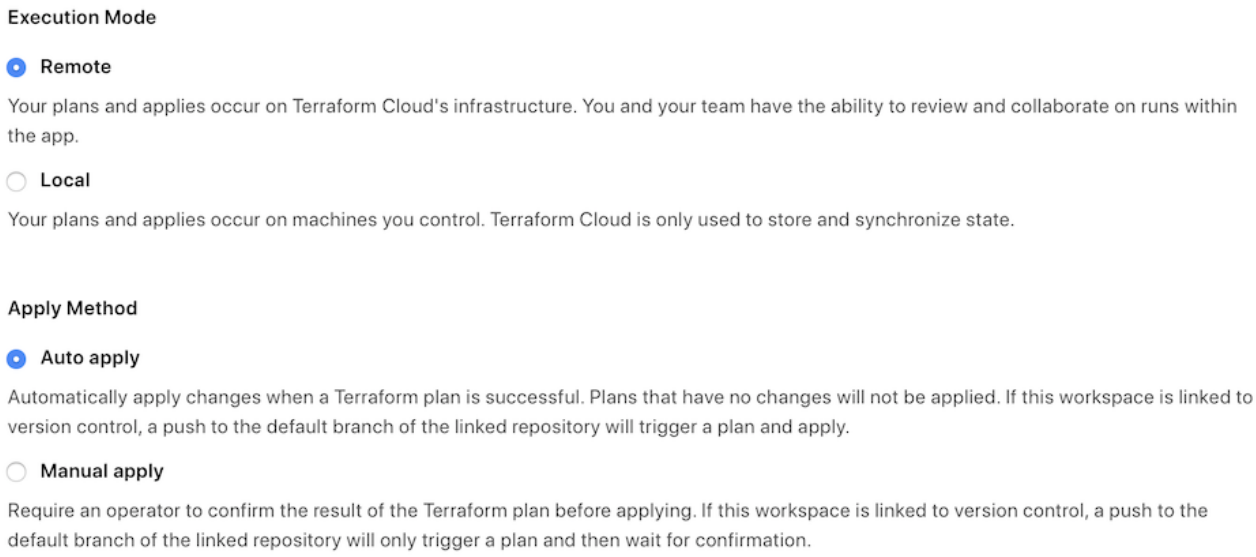
Using Github for VCS and Triggering Plan and Apply Functions

I've setup this workspace to a specific VCS configuration and I want every change that occurs in the configuration to trigger a new plan. There are options to only watch a specific file but my use-case is specific that I am looking for lots of potential configuration changes so I'm just monitoring the entire folder.



This means that every `git push` that is run will trigger Github to send a webhook back to TFE to run a plan, and if any state changes are needed, to run the apply process.

You can choose to require confirmation or to auto-apply. For now I'm going to have it wait for confirmation. This workspace is also setup for notifications using Slack so the team all get notifications when plans are run and if any interaction is needed.



The Slack integration is super easy to use and this ensures that anything happening is known across the team in real-time in addition to the usual state management and activity which is logged continuously inside Terraform Enterprise.

Setting Up Your Shell Environment

To run the scripts included in the repository requires an API key for Terraform and Github. This is defined in the script as the following variables:

```
$TFE_TOKEN
$GH_TOKEN
```

I use a local file called exports.sh which creates the environment variables for me before the run. If either token is missing, the API calls will fail so you'll pretty quickly realize when something is not right.

Updating the Instance Size Using the Terraform Enterprise API

There are two steps we must use to update the TFE input variable. First we have to find out the variable ID based on the name of the input variable and then that is embedded in JSON data to pass to the TFE API again to update the variable.

All of the code for this is inside a Bash script which is `tfe-t8c-update.sh` so that you can run from a shell easily and pass the mandatory parameters including the TFE organization name, TFE workspace name, and instance size variable.

The command structure in my case would be:

```
sh tfe-t8c-update.sh Turbonomic tfe-turbonomic-demo t2.small
```

This script runs and you'll get some JSON output confirming the change. There is no notification otherwise so you have to look at the JSON output or your Terraform Enterprise configuration to see the changes.

Triggering a Terraform Enterprise Plan by Using the Github API

If you're like me, you probably think that you should just be able to kick off a plan directly by using the TFE API but there are actually multiple steps needed to do that and it requires uploading your TF configuration files in order to queue the plan/apply. I worked with the HashiCorp support folks and confirmed that this is by design because VCS-connected workspaces always want to confirm the code is current and that requires a push from Github rather than a pull from TFE.

This is where we make use of the Github API. I need to update a file in the repository just like I would in a `git push` but I don't want all the repo code locally to have to do this and run a CLI to add+commit+push process. Thankfully the API makes it wicked easy for setting up a trigger file to do this right in the Github repo.

I've got a file literally named `trigger` in the root of the repo which I'm going to updated the contents of each time I run the process. It's just a simple time stamp. For the content of the file we need to use Base64 encoded content so this how we generate our timestamps:

```
trigger_content=$(echo date | base64)
trigger_date=$(date +'%a%b%d-%H%M')
```

To update the file using the Github API we first need to find the SHA signature to enable us to update the existing file. This is done dynamically so we always know we are using the latest revision. The `$github_trigger_url` will be where your file is located. I've added mine right into the public repository so you can see the actual end-to-end script.

```
# Get the current trigger file SHA from Github
trigger_sha=$(curl -s \
--header "Authorization: token $GH_TOKEN" \
--request GET \
$github_trigger_url \
| jq '.sha' | sed -e 's/^"//' -e 's/"$//')
```

This gathers the file info, passes the output to JQ which is a wicked cool JSON parser, then strips off the opening and closing quotation marks using the sed command.

Now that we have set the `$trigger_sha` variable, we can create the JSON data needed to send to the update API request:

```
git_json_data={\""message\"":\""Updated by Turbonomic
$trigger_date\"",\""committer\"":{\""name\"":\""Turbonomic
Labs\"",\""email\"":\""eric@discoposse.com\""},\""content\"":\""$tr
igger_content\"",\""sha\"":\""$trigger_sha\""}
```

This is used to now pass to the final API call we make at the last section of the `git-trigger.sh` file:

```
curl -s \
--header "Authorization: token $GH_TOKEN" \
--request PUT \
--data "$git_json_data" \
$github_trigger_url
```

Phew! We made it! That is the last step in the process which will update the trigger file in the Github repo with a commit message. That also causes the automatic send of a webhook back from Github to Terraform Enterprise through the native VCS connection.

Why Do it This Way?

This is a pre-cursor to working on the integration with Turbonomic (or any external system) that needs to be able to dynamically update the live infrastructure while also ensuring the Terraform configuration is kept up to date. Keep watching for more to come on expanding this use-case and the Turbonomic side of the connection to put this fully into place.

Demo in Action!

Here is the quick walkthrough of exactly what the end-to-end process looks like. Hopefully this gives a good context to the flow and you can always drop in a comment to ask any questions about updates or ideas you want me to explore.

Helpful articles for the API usage in this article:

- Github repository contents API – https://developer.github.com/v3/repos/contents/
- Terraform Enterprise Variables API – https://www.terraform.io/docs/cloud/api/variables.html