Mike Ensor

Follow 111 Followers About

Trouble with eventual consistency, Terraform and Google Cloud

Mike Ensor Jul 24, 2019 · 5 min read ★

Over the last few days I've been building out a multi-tiered mock enterprise reference solution based on my speaking engagements and several projects I have built over the last 3–4 years. In my pursuit of this ambitious project, I ran into a problem when building a Terraform `module`. All of the resources I have built are synchronous and blocking when using Terraform to provision them, with the exception of `google_project_service` and `google_project_services`.

Google requires users to enable specific APIs to manage resource lifecycle in addition to standard IAM. These APIs need to be enabled to create `resources` like `google_container_cluster` instances over the Google Cloud API. Several of the APIs require extra steps including agreeing to terms and conditions or creating credential

I strive to make all of my (and my client's) projects 100% code-based solutions. With the introduction of API-driven cloud-based or modern virtualized infrastructure, coupled with Infrastructure-as-Code abstraction tools like Terraform, Cloud Formation, Build Manager and Pulumi, architectures should strive to be 100% code based. The benefits are numerous and I do intend on going into the benefits, so here are a few blogs if you need to be convinced: https://medium.com/@timhberry/infrastructure-as-code-but-why-ab13951fb8d4 , http://techtowntraining.com/resources/blog/infrastructure-as-code-benefits (sorry, this has a big full-page ad) and https://youtu.be/eiHgx-pyg1U (a shameless plug on me talking about the 'no-ops culture' and designing architectures using Infrastructure-as-Code).

The problem I ran into is around a code side-effect or asynchronous actions or eventually consistency (likely the latter). Google API enablement needs time to propagate around their solution. My plan is to to dynamically build a `google_project` and then create a GKE (Kubernetes) instance inside the newly created project using a Terraform `module` I had previously created.

```
## main.tf

resource "google_project" "project" {
  name            = "testing-project"
  project_id      = "project-${local.project_suffix}"
```

```
  labels           = { "example" = "true" }
}

module "gke" {
  source                 = "../modules/gke"
  project_id             = "${google_project.project.project_id}"
  region                 = "${var.region}"
  location               = "${var.location}"
  region_zone            = "${var.region_zone}"
  credentials_file_path = "${var.credentials_file_path}"
  cluster_prefix         = "cluster-test"
  max-node-count         = 5
}
```

The above code would fail due to the `container.googleapis.com` not being enabled. The fix should be simple, just add code required to enable the API.

```
...
... # still in main.tf

resource "google_project_service" "gke-api" {
  project = "${google_project.testing-project.project_id}"
  service = "container.googleapis.com"

  disable_dependent_services = true
  disable_on_destroy         = false
}
```

All good, right? No…FAIL

The Terraform validates and runs, but when the `module` attempts to create a GKE cluster using `google_container_cluster` the build fails with a `403 Forbidden` error, despite the API enablement code succeeding. If I re-run the build, only seconds later, the code succeeds. This violates one of my strong beliefs in the principles of Continuous Delivery, every build should be predictable.

The problem is with the API needing to be propagated across Google's solution. The underlying Google API used by Terraform returns with "success", but the API is not consistent across the platform. With normal resources, the problem would be solved using `depends_on` keywords that give Terraform information about how resources may be connected if the syntax does is not clear or the associations are not explicit. However,

I tried several different solutions including `external` data providers and `null_resources` in order create some sort of waiting time allowing the propagation to take place. I ended up using a four-step plan that can be used for just about any Terraform script that ends up with eventual consistent resources despite the API result being "complete".

My solution

1. Setup the API (or eventual consistent resource) inside of the `module` (this is key)
2. Create a `null_resource` instance that contains a simple script to `sleep` (NOTE: this solution only works for linux-based solutions).

```
# module/gke/main.tf
...
resource "null_resource" "resource-to-wait-on" {
  provisioner "local-exec" {
    command = "sleep ${local.wait-time}"
  }
}
...
```

3. Have the `null_resource` code depend on the output of the `google_project_service` resource previously created. Note, I have created a variable `local.wait-time` and set the value to "60", representing 60 seconds.

```
# module/gke/main.tf
...
resource "null_resource" "resource-to-wait-on" {
  provisioner "local-exec" {
    command = "sleep ${local.wait-time}"
  }
  depends_on = ["google_project_service.gke-api"]
}
```

`depends_on` clause to the resource forcing Terraform to wait until the `null_resource.resource-to-wait-on` has completed.

```
# module/gke/main.tf
...
resource "google_container_cluster" "primary" {
  name     = "${var.cluster_prefix}-${local.cluster_suffix}"
  location = "${var.location}"
  project  = "${google_project.testing-project.project_id}"

  ..
  .. details removed for brevity
  ..

  depends_on = ["null_resource.resource-to-wait-on"]
}
```

The above pseudo-code represents what I have used inside of my module. I pass the `project_id` into my module and a few other required fields. Now the Terraform waits for the calculated amount of time before trying to create the `google_container_cluster` instance.

Conclusion

First, I really wish I did not have to create a #hack, but the rate at which cloud providers are adding functionality, and the rate at which Hashicorp (and community) are keeping up features to the new features, it is reasonable that we will need to have little hacks from time-to-time. With one final note, please make sure to keep these “sleep” hacks to a minimum, and only use them IF there is no alternative. Terraform has a declarative format that allows the underlying algorithms to achieve time and resource efficiency. Adding 60 seconds here or there can drastically change the build time for new stacks.

Terraform Google Cloud Platform Eventual Consistency Hacks Infrastructure As Code