Terragrunt — The glue around Terraform

Patrick Picard
Nov 16, 2020 · 4 min read

Terragrunt Logo

It's been a while since I've posted and it is primarily because I have been super busy on a recent engagement. One the challenges we faced was orchestrating the deployments of multiple Terraform stacks.

Terraform excels as a Infrastructure as Code DSL. However, it lacks when it comes to building an environment where deployments are interdependent. This becomes very challenging when you want to manage all the infrastructure for a landing zone (foundational deployment of core infrastructure for a cloud platform).

Before going further, let me clarify some of the terms I will be using:

- Module — Terraform modules encapsulate logic to deploy a component of a stack. A stack can have multiple modules to make the solution. Multiple modules can be connected together in a larger module to build a stack. For example, you can combine a storage account, resource group, managed identity and a Virtual Machine in a larger module that is used to create an Azure DevOps agent.

- Stack — Multiple cloud resources that provide a bounded capability. A stack should have its own state file. Examples: core network, DNS zones, DNS forwarders, management group hierarchy, centralized logging. Often times, you will deploy a stack multiple times for each environment (dev, qa, prod)

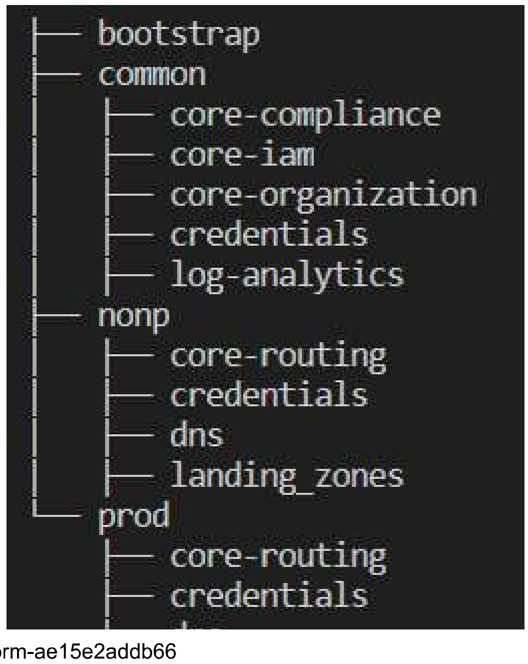
Terragrunt ‘s scope of deployment is a terraform module (the larger module defined above). It is capable of stitching multiple stacks using dependencies.

Initially, I looked at using remote states to share information between deployments. However, additional concerns remained to manage the environment at scale (I’ll get to those shortly). I continued my search and ran into a tool from Gruntwork called Terragrunt. After a quick POC (3 hours hacking things together), I was convinced this tool had legs. Let’s go over what Terragrunt is.

> *Terragrunt is a thin wrapper that provides extra tools for keeping your configurations DRY, working with multiple Terraform modules, and managing remote state*

Let’s dive into the capabilities. I plan on having additional blog entries to cover them individually.

- **Hierarchical structure** — Deploying components of an infrastructure is hierarchical in nature. Organizing those components in code follows a similar approach. This allows you to configure your terragrunt code with a hierarchy and inherit configuration from higher levels in the tree; thus following DRY.

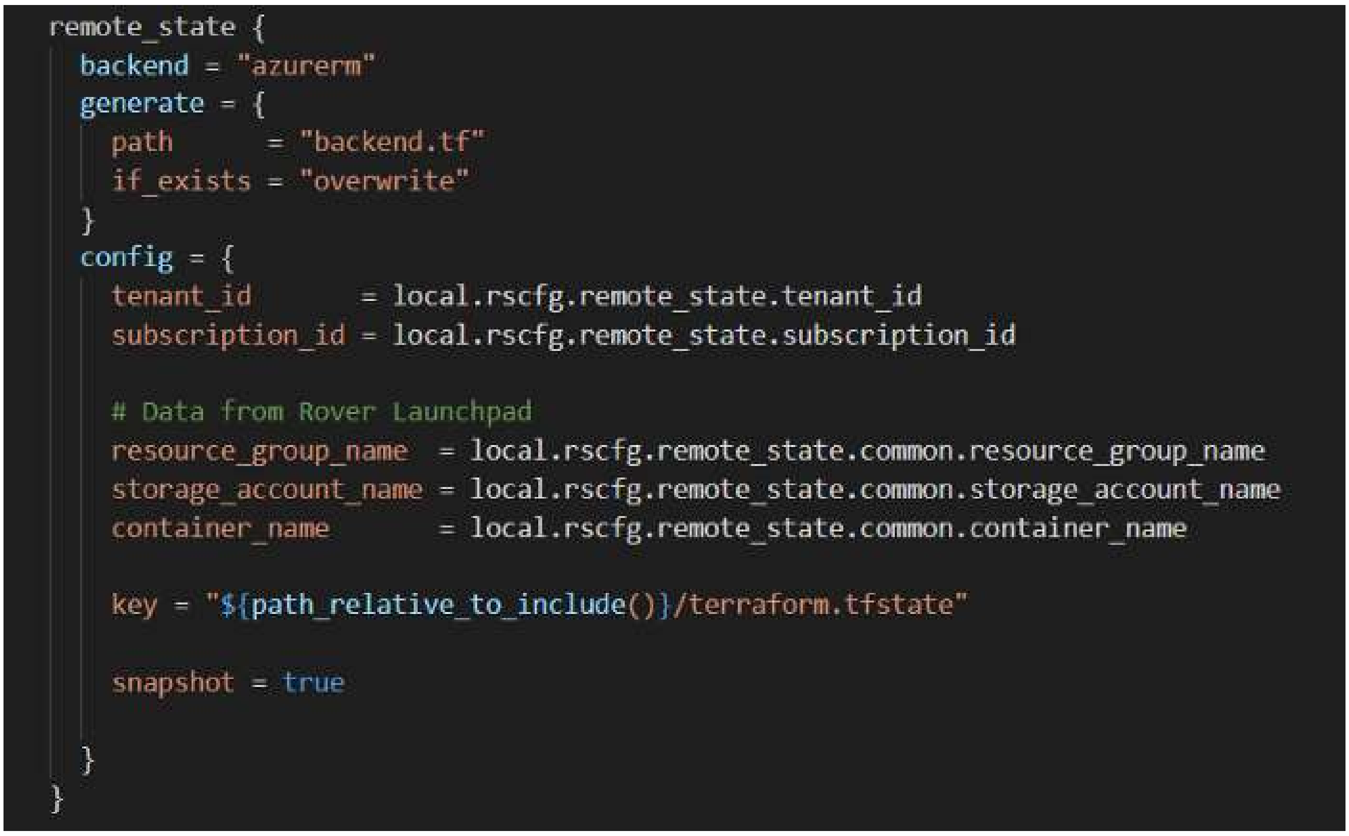


```
├── bootstrap
├── common
│   ├── core-compliance
│   ├── core-iam
│   ├── core-organization
│   ├── credentials
│   ├── log-analytics
├── nonp
│   ├── core-routing
│   ├── credentials
│   ├── dns
│   ├── landing_zones
└── prod
    ├── core-routing
    ├── credentials
```

```
    ├── dns
    ├── landing_zones
```

Terragrunt hierarchical structure

- **Dynamic Remote State Management** — Let's imagine you have 10 components to manage and two instances of those components to cover production and non-production. You now have to manage ~20 state files. Therefore, you need to create a backend configuration for each. This becomes unwieldy quickly. Terragrunt provides the ability to generate backend configurations on the fly using a *remote_state* resource and creating the remote state key using functions.

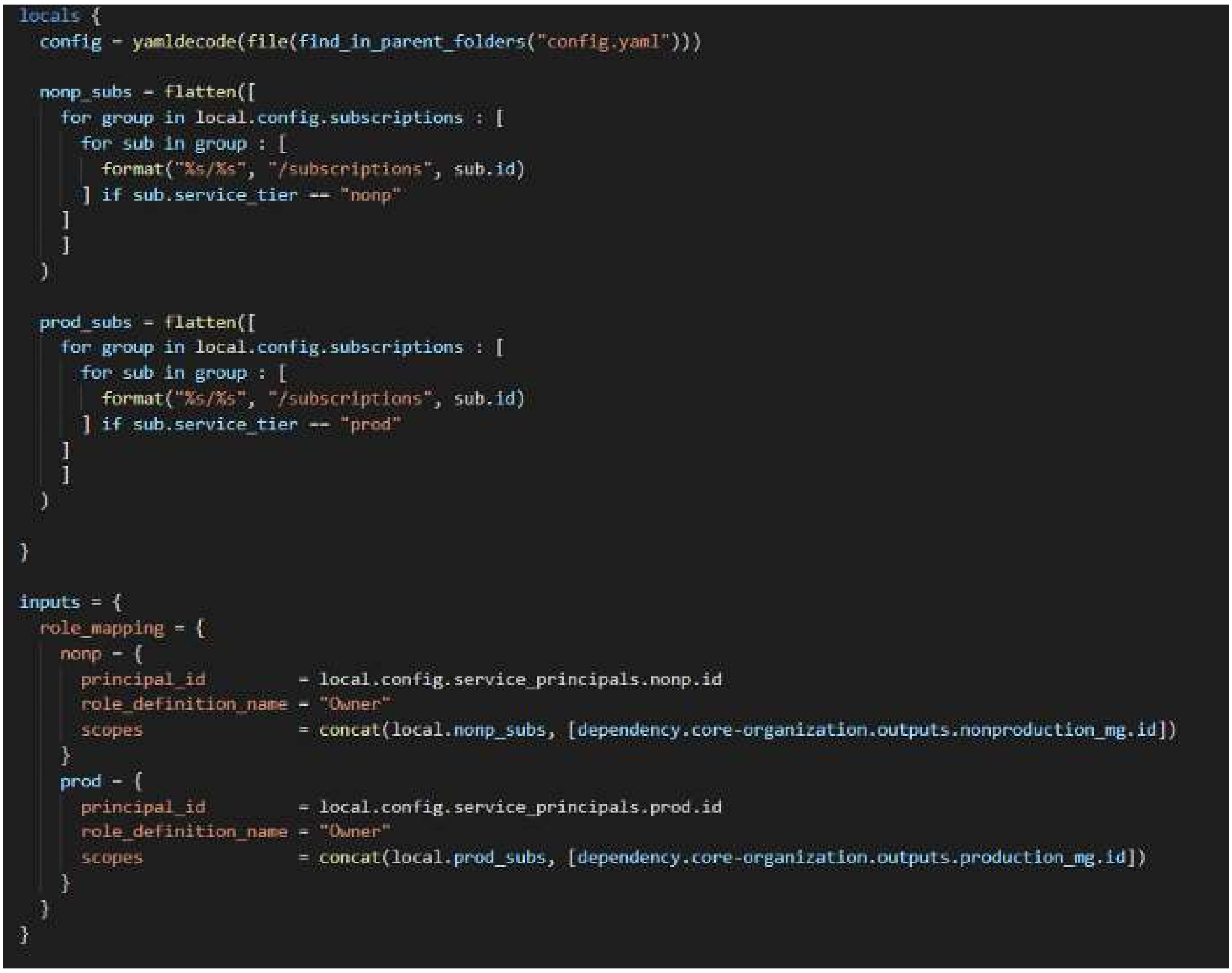


```
remote_state {
  backend = "azurerm"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }
  config = {
    tenant_id       = local.rscfg.remote_state.tenant_id
    subscription_id = local.rscfg.remote_state.subscription_id

    # Data from Rover Launchpad
    resource_group_name  = local.rscfg.remote_state.common.resource_group_name
    storage_account_name = local.rscfg.remote_state.common.storage_account_name
    container_name       = local.rscfg.remote_state.common.container_name

    key = "${path_relative_to_include()}/terraform.tfstate"

    snapshot = true

  }
}
```

Remote State Dynamic Generation

- **Full DSL expressiveness when passing values into a module** — Variables provide a contract to your module. With vanilla Terraform, the values must be calculated ahead of time and are not dynamic. With terragrunt, you have full access to the Terraform language when "calculating" inputs to a module; thus significantly reducing the workarounds to generate the data outside of Terraform.

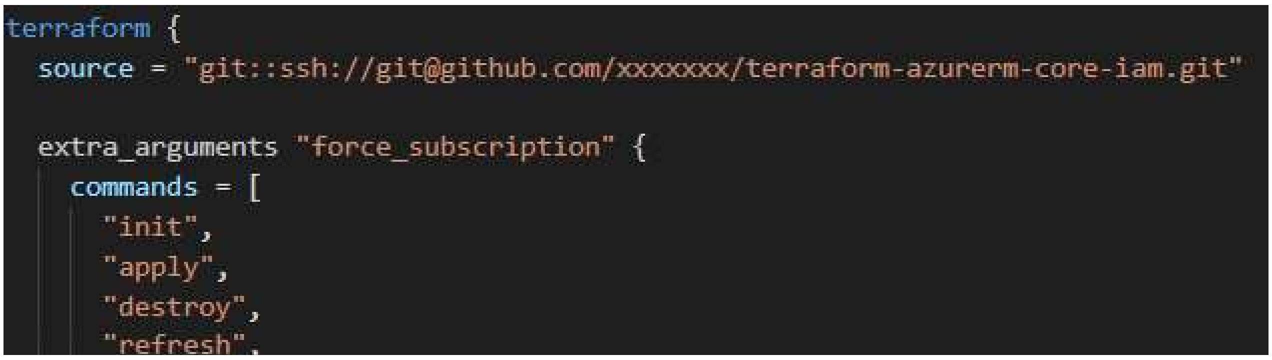
- **Inter-module dependency management** — Let's face it, Terraform is basically a graph engine to determine the order of deployments for resources within a module. Terragrunt takes this graphing engine to the next level by providing graph/dependency management between modules/stacks. It allows to pass output data from one module to the next using dependency resources.



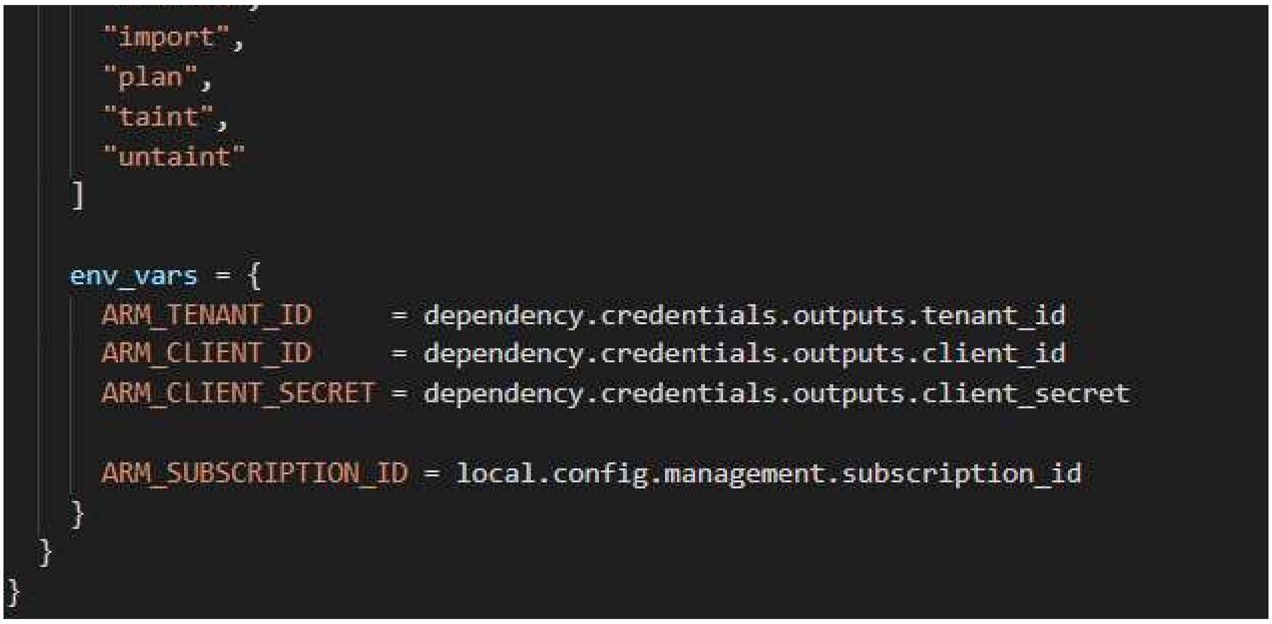
```
locals {
  config = yamldecode(file(find_in_parent_folders("config.yaml")))

  nonp_subs = flatten([
    for group in local.config.subscriptions : [
      for sub in group : [
        format("%s/%s", "/subscriptions", sub.id)
      ] if sub.service_tier == "nonp"
    ]
    ]
  )

  prod_subs = flatten([
    for group in local.config.subscriptions : [
      for sub in group : [
        format("%s/%s", "/subscriptions", sub.id)
      ] if sub.service_tier == "prod"
    ]
    ]
  )

}

inputs = {
  role_mapping = {
    nonp = {
      principal_id         = local.config.service_principals.nonp.id
      role_definition_name = "Owner"
      scopes               = concat(local.nonp_subs, [dependency.core-organization.outputs.nonproduction_mg.id])
    }
    prod = {
      principal_id         = local.config.service_principals.prod.id
      role_definition_name = "Owner"
      scopes               = concat(local.prod_subs, [dependency.core-organization.outputs.production_mg.id])
    }
  }
}
```

DSL Expressiveness & Dependency Management

- **Multi-subscription/Account support** — Terraform is primarily focused on deploying components to a single scope such as an Azure Subscription or an AWS Account. It is possible to create aliases in the provider configuration to get around this, but this becomes unmanageable VERY quickly. Terragrunt allows the generation of environment variables to pass into Terraform CLI; thus allowing multi-subscription support.



```
terraform {
  source = "git::ssh://git@github.com/xxxxxxx/terraform-azurerm-core-iam.git"

  extra_arguments "force_subscription" {
    commands = [
      "init",
      "apply",
      "destroy",
      "refresh",
```



```
      "import",
      "plan",
      "taint",
      "untaint"
    ]

    env_vars = {
      ARM_TENANT_ID     = dependency.credentials.outputs.tenant_id
      ARM_CLIENT_ID     = dependency.credentials.outputs.client_id
      ARM_CLIENT_SECRET = dependency.credentials.outputs.client_secret

      ARM_SUBSCRIPTION_ID = local.config.management.subscription_id
    }
  }
}
```

Multi-Subscription Management

- **Terraform Code Generation** — Terragrunt is able to generate configuration files on the fly during the plan/apply/destroy phases. This is useful to generate backend configurations. Similarly, you can use the *generate* resource to create provider configurations. By generating configurations, you are able to reduce code duplication and follow DRY principles (don't repeat yourself).
- **Multi-component plan/apply/destroy** — Terraform focuses on a single stack deployment. Terragrunt expands this to work with multiple stacks based on the dependencies defined in the Terragrunt code. You can do plan-all (not perfect, more in a future article), apply-all, and destroy-all. Terragrunt builds the meta-graph and will apply it in the appropriate order.
- **It just works** — OK, that is not a capability. But, developer experience is paramount and Terragrunt really nailed it by keeping the DSL Terraform-like. If you are comfortable with Terraform, expanding skills to Terragrunt is super easy to grasp.

So while some of those capabilities may sound cryptic, stick around in this blog series around Terragrunt. I will break down each capabilities with code examples.

If you are curious in the mean time, have a look at https://terragrunt.gruntwork.io/docs/

Terraform Terragrunt Azure