Azure DevOps Pipelines: Leveraging Stages with YAML Objects

Introduction

By now we've covered the basics on tasks, jobs, stages, environments, variables and templates for jobs and tasks. It is now time to move towards stage templates. Additionally, to assist with templating we will cover one way to leverage YAML objects which is a key tool when templating up and keeping your pipelines DRY (Don't Repeat Yourself).

To follow along I am leveraging non-ADO code in from Azure IaC Flavors and YAML templates from TheYAMLPipelineOne

Stages

Recapping from a pervious post, a stage is:
Stages are the level above jobs. A stage is required; however, there is a default if one is not provided. A stage can contain one to many jobs and acts as a logical grouping of related jobs.

An important component to understand is that Azure DevOps, out of the box, provide an easy way to rerun Pipelines at the stage level. This means our stages should be constructed with jobs that make sense to re-run together. Usually, customers want to naturally associate a stage with a customer environment. I'd advise fighting this temptation.

Picking up on this notion, I tend to recommend a stage can be combination of app components (data, front end, infrastructure, etc..) and deployment target (region, virtual machine, etc..). By making this distinction it helps keep our stages scoped to an easily and targeted deployment path.

When stopping to think about this our stages, specifically between a test and production environment, should be exactly the same minus the deployment target information and environment settings. Thus, they are a great candidate for templates.

So why weren't these covered in jobs and task templates in the previous article? Stages are the container that houses all of the deployment jobs. As such extra thought should be placed into how to architect these to scale out infinitely. It shouldn't matter if I am deploying to one or twenty locations. They all should contain the same jobs, tasks, and appropriate variables. Deploying to one stage or two stages with the same template is easy....but how to do it for infinite possibilities? Enter YAML Objects.

YAML Objects

To Clarify YAML Objects can be used with more than just stages. Previously I have used YAML Objects to pass project specific configuration into a stage deploying multiple projects as one

such example. Where each project has it's own specific information and the object is passed into a subsequent job. If you read through Azure DevOps documentation, there is mention of YAML Objects; which now includes an example. If wanting more focused on the topic check out my personal blog post on YAML Objects and Looping.

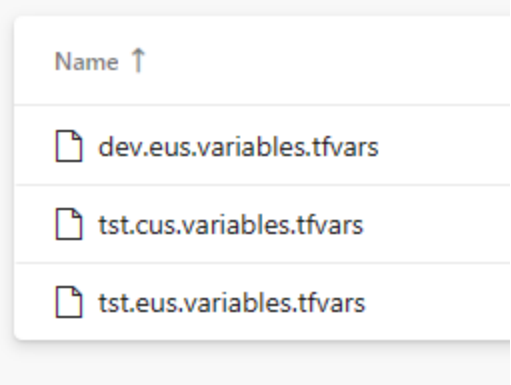
For the purposes of this article, I am going to keep this as simple as I can. We will deal with just deploying Terraform to multiple environments and multiple regions. My warning is a better practice would be to deploy Infrastructure as Code and any application code together. There are strong opinions of this; however, I am of the camp you shouldn't have one without the other so they should be stored, managed and deployed together.

Here is my parameter which will be passed into the stage template:

```yaml
parameters:
- name: environmentObjects
  type: object
  default:
    - environmentName: 'dev'
      regionAbrvs: ['eus']
    - environmentName: 'tst'
      regionAbrvs: ['eus', 'cus']
```

Essentially, we have an environment, this will match the environment name defined in Azure DevOps. I can't state this enough that a consistent naming pattern across all aspects gives you the freedom to define one variable and build the rest.

I even take this down to the Terraform variable file names:



A quick disclaimer as someone keen would say why you need region-specific parameters? Good question, in this case assume you may have a primary region that has different load requirements than a secondary.

The important concept here with working with templates is passing in the YAML Object to the stage template. The logic for looping and creating all the individual stages is actually handled by the template. Remember that the YAML pipeline will fully expand when submitted to Azure DevOps for execution.

To call the stage template will look like:

```yaml
- template: stages/terraform_apply_stage.yml@templates
  parameters:
    environmentObjects: ${{ parameters.environmentObjects }}
    serviceName: ${{ parameters.serviceName }}
```

serviceName name is used to help define the Terraform state file as well as provide names to the stages and jobs. The key thing to recognize here is the passing of the entire environmentObjects block. This will include the configuration for all the environments and regions. The next step is to take that information and loop through the environmentObjects data creating the appropriate stages. For this purpose, keep in mind we need to loop not only for environments; however, regions as well.

```yaml
stages:
  - ${{ each environmentObject in parameters.environmentObjects }} :
    - ${{ each regionAbrv in environmentObject.regionAbrvs }} :
        - stage: '${{ parameters.serviceName }}_${{ environmentObject.environmentName}}_${{regionAbrv}}_tf_apply'
          variables:
            ${{ if  ne(environmentObject.dependsOn,'')}} :
                dependsOnEnv: '${{ parameters.serviceName }}_${{ environmentObject.dependsOn}}_${{regionAbrv}}_tf_apply'
            ${{ else }} :
                dependsOnEnv: '${{ parameters.serviceName }}_tf_build'
          dependsOn: ${{variables.dependsOnEnv}}
          jobs:
          - template: ../jobs/terraform_apply_env_job.yml
            parameters:
              environmentName: ${{ environmentObject.environmentName }}
              serviceName: ${{ parameters.serviceName }}
              additionalParameters: ['-var-file="variables/${{ environmentObject.environmentName }}.${{regionAbrv}}.variables.tfvars"']
```

Something to consider is the `stages` keyword appears in both the template and the calling pipeline. This true of any templates and I link to think of it as the key word marker that the template will load into.

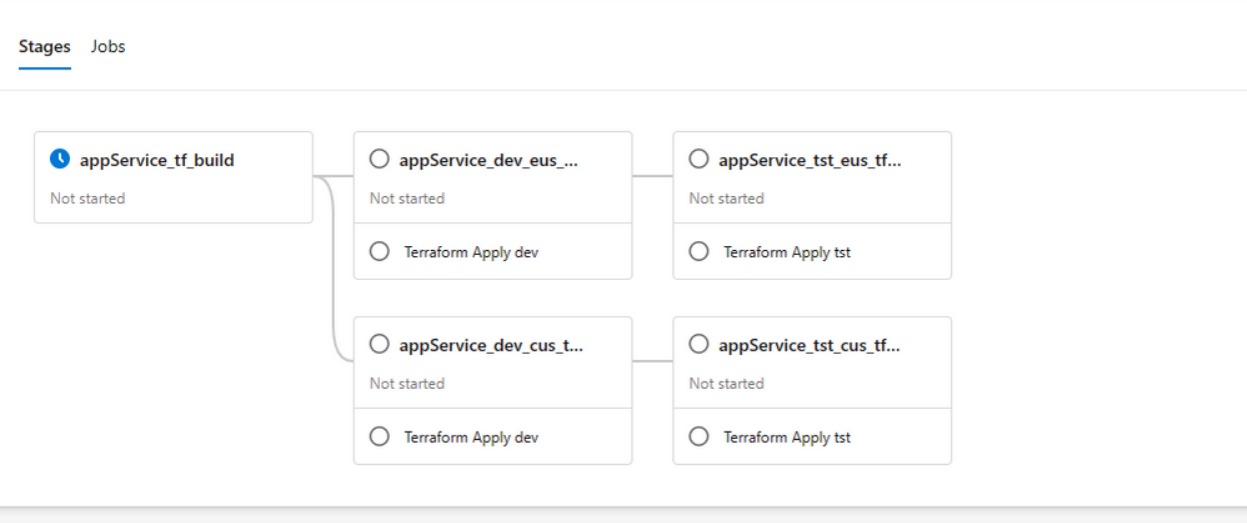
The first loop here is to go through all the environmentObjects, in this case ‘dev’ and ‘tst’. Once inside the instance of the object need to loop through all the regionAbrv values in the regionAbrv array. Notice though that these each loops are being called prior to the actual stage creation. This is the key to making the number of stages dynamic. Since YAML will fully expand at execution it will determine and pull in the appropriate stages at that time.

Something to consider when dealing with any type of dynamic template insertion is the stage name must be unique across the pipeline and the jobs must be unique in the same stage. Because of this it is important to build a name that relies on the individual components of the loop. This ensures that there is no class of two stages or jobs having the same name.

The next step here is using logic to determine variable assignment. I realize this is a bit tricky and honestly probably one of the more confusing parts on this post. The thought process behind this is if we are fanning out to x number of stages then we need to be able to track dependencies. By default, stages are dependent on each other in sequential order.

Well in this scenario I'd like all the tst jobs depend on completion of all the dev jobs. Jobs in dev should realistically be dependent on the build stage, which in this case due to a strict naming convention we know what the name of that build stage will be. As such if no dependency is being passed in then we will assign the build stage as the dependency. There will be a follow up post on using if statements in YAML pipelines later in the series.

Once we get through this our pipeline execution plan will look like:



Conclusion

This is not an easy topic and deals with a lot of advanced concepts when not only creating your pipelines; however, even more when starting to think how templatize them. If done properly the possibilities of scale can be limitless.