Azure DevOps Pipelines: Practices for Scaling Templates

Introduction

This article is part of a larger series on Azure DevOps Pipelines regarding leveraging YAML templating. As part of this series, I have had numerous requests or questions around what is considered "good practices" when looking to leverage one repository consisting of YAML templates.

Objective

The objective on what we are trying to achieve is create a structure by which we can create a task, job, stage, or variable template once and let all of our pipelines reuse the code. The intent is to achieve the following outcome:

- Reduce pipeline creation time
- Ease maintenance of pipelines
- Provide an extra layer of security on who can create pipelines
- Consistently be able to deploy various apps in a similar fashion

Template Expansion

To best understand and architect a YAML pipeline via templates have to understand how the template expansion process works. When we go to run our Azure DevOps Pipeline it will collect all the YAML templates and associated files, variables, etc... and expand it into ONE file. This is the azure-pipeline-expanded.yml file which can be downloaded post a job completion. This is important to understand from architecture and troubleshooting perspective that we can see that one file is submitted to Azure DevOps. All parameters being passed in must be available at the pipeline compilation time. Runtime variables can be runtime variables and will be retrieved at pipeline execution.

One Repository

In order to help achieve these goals it is recommended to store of the templates in a dedicated Git repository. Azure DevOps pipelines support this functionality via the resources block. By doing this all of our pipelines can retrieve their templates from one consolidate location. In addition, by putting these in a separate repository we can introduce another layer of security on who can update this repo.

Connecting to Single Replository

When connecting to a single YAML repository one just needs to declare the repository reference like:

```yaml
resources:
  repositories:
  - repository: templates
    type: github
    name: JFolberth/TheYAMLPipelineOne
    endpoint: JFolberth
```

Let's break down what each one of these actually means.

```yaml
resources:
    repositories:
    - repository: string # Required as first property. Alias for the repository.
      endpoint: string # ID of the service endpoint connecting to this repository.
      trigger: none | trigger | [ string ] # CI trigger for this repository, no CI trigger if skipped (only works for Azure Repos).
      name: string # repository name (format depends on 'type'; does not accept variables).
      ref: string # ref name to checkout; defaults to 'refs/heads/main'. The branch checked out by default whenever the resource trigger fires.
      type: string # Type of repository: git, github, githubenterprise, and bitbucket.
```

Template Scope

When creating templates for reuse one of the first thing one must decide is what is the scope of the template file? It is recommended that a template have a predefined scope and designed for one activity. At the most rudimental level a task template will consist on one task. No more. This ensures each individual component can be separately maintained and used when needed. When needing to do more then one task then we create a template for one job. One job will call one or multiple tasks. This ensures we wrap related tasks together. If multiple jobs are required, then we create a template for one stage that calls multiple job templates.

Let's walk through a practical example of this leveraging the common steps of creating a build stage for a building a dotnet app:

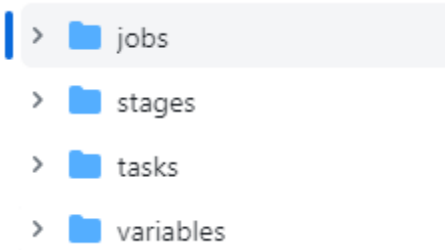
- ensure proper dotnet SDK
- build dotnet app via DotNeCoreCLI
- publish dotnet app via DotNetCLICLI
- publish dotnet code via PublishPipelineArtifact

These tasks would be chained together via a job and that job then would be contained in a stage. I recommend a separate file for each. That means if you are keeping count, we'd have one stage template file calling one job template file which calls four individual task template files.

If your deployment footprint is small this may seem like a decent amount of overhead. However, I implore you to focus on just the PublishPipelineArtifact. This one task really is critical to any pipeline to publish an artifact as part of the pipeline execution. By templating it we can now insert this task into anyone pipeline. Even better say Microsoft updates the version of the task from v1 to v2. We can now change it in one file via one Pull Request and instantly all of our consuming pipelines get the updated.

File Structure

It is recommended that there is a folder per template type. This means that a consolidate repository for YAML templates would most likely look something like:



If leveraging the intent that each template contains one instance of a stage, job, task, variable file this quickly becomes organized.

Naming Convention

Each organization may adopt this differently and this is just the pattern I have found to be successful. I typically name my YAML templates some format of technology_verb_scope.yml.

I have found this helps with quickly identifying what technology the template is for, what it does, and lastly what it should be used for.
Here is a quick example of some:
adf_build_job.yml = Azure Data Factor Build Job Template

zip_publish_job.yml = Zip files and publish them as an ADO Artifact Job Template
bicep_deploy_stage.yml = Deploy Bicep files Stage Template
azpwsh_file_execute_task.yml = Azure Power Shell execute a file Task Template
node_install_template_task.yml = Install Node on the server Task Template
azure_dev_variable.yml = Variables to be used for an Azure Dev Environment Template

One thing I have learned the hard way is the file names **are case sensitive.** This is important and due to this I recommend just leaving them a lower case to avoid any confusion.

Calling a template in a remote repository

Once we have the connection to the remote repository defined, we need to be able to tell Azure DevOps what file to connect to.

Here is an example where information is being passed into a bicep build stage template:

```
parameters:
- name: environmentObjects
  type: object
  default:
    - environmentName: 'dev'
      regionAbrvs: ['eus']
    - environmentName: 'tst'
      regionAbrvs: ['eus']
- name: templateFileName
  type: string
  default: 'main'
- name: templateDirectory
  type: string
  default: 'Infrastructure'
- name: serviceName
  type: string
  default: 'adfdemo'

stages:
- template: stages/bicep_build_stage.yml@templates
```

```
  parameters:
    environmentObjects: ${{ parameters.environmentObjects }}
    templateFileName: ${{ parameters.templateFileName }}
    serviceName: ${{ parameters.serviceName }}
    templateDirectory: ${{ parameters.templateDirectory }}
```

Notice that we tell Azure DevOps we want the stages/bicep_build_stage.yml file located in what we have called @templates. This is what we put in for the 'repository' value when defining the reference. This is an alias that only our local pipeline is aware of.

Calling templates within templates

When opening up the bicep_build_stage.yml file the template above is using we can see it will call a job template:

```
parameters:
- name: environmentObjects
  type: object
  default:
    - environmentName: 'dev'
      regionAbrvs: ['cus']
- name: templateFileName
  type: string
  default: 'main'
- name: templateDirectory
  type: string
  default: 'Infrastructure'
- name: serviceName
  type: string
  default: 'SampleApp'
```

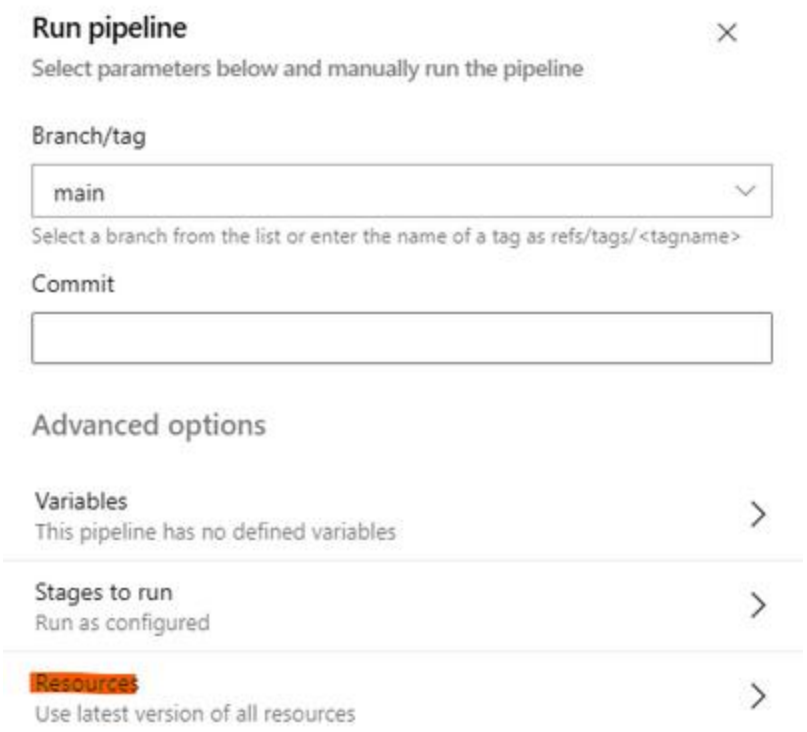
```yaml
stages:
- stage: '${{ parameters.serviceName }}_build'
  jobs:
  - template: ../jobs/infrastructure_publish_job.yml
    parameters:
      targetPath: ${{ parameters.templateDirectory }}
  - ${{ each environmentObject in parameters.environmentObjects }} :
    - ${{ each regionAbrv in environmentObject.regionAbrvs }} :
      - template: ../jobs/bicep_whatif_env_job.yml
        parameters:
          environmentName: ${{ environmentObject.environmentName }}
          templateFileName: ${{ parameters.templateFileName }}
          templateDirectory: ${{ parameters.templateDirectory }}
          serviceName: ${{ parameters.serviceName }}
          regionAbrv: ${{ regionAbrv }}
```

First notice when calling a nested template, we don't need the @templates. This is important as if we include this reference in here then our pipeline is dependent on finding that repository alias...even if we named it something different in our calling pipeline. The second thing to note here is we are calling the RELATIVE PATH of the YAML template repository. In our case this means up a folder from the stage folder and then locate the bicep_whatif_env_job.yml file under the jobs folder.

Running Different Template Versions

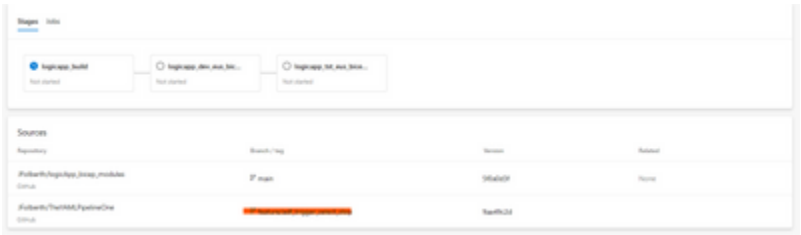
Perhaps you are making some changes and updates to the master template repository and need to test them out without impacting your production version. To accomplish this Azure DevOps let's you chose an easy way to change the branch version of your required resource templates.

Within Azure DevOps select the pipeline to run and select Resources:



This will load the repository resources in our resource block. Select the one your templates are located in. From there we can select which branch to use:

When our pipeline runs, we can confirm the branch that is being used.



Alternatively, we could also update the repository reference in our calling pipeline to point to a specific branch if it is more than a few minor changes.

When to use Conditions and When to use Expressions

This is from [Azure DevOps Pipelines: If Expressions and Conditions] and I am including here for consolidation of practices.

Conditions should be leveraged when requiring to evaluate the status of something that has been ran or loaded into the template. Remember that if expressions will dynamically insert templates or variables into a pipeline.
This means if expressions can only evaluate information that is static and available at time of task/job/stage execution. They will not know about which jobs have succeeded, failed or unaware of any variables that may have been created as part of a proceeding task/job/stage.

My own personal pattern is to default leveraging if expressions first. This allows for a cleaner UI and a simpler approach when managing pipelines. This is due to only loading the necessary information into the pipeline vs load everything and evaluate as it goes. The flipside; however, is more complicated pipelines may require additional conditional operators and thus the condition attribute is more appropriate.

Running Jobs in Parallel

By default, Azure DevOps Pipeline jobs will run in parallel. This is extremely powerful and significantly impacts how jobs are structured. Think of a job as an isolated list of related steps to accomplish an end goal. There are times where we want to leverage parallel activities in the hopes of speeding up our CI/CD pipelines.

One such example I commonly recommend is around Terraform. When running a Continuous Integration (CI) build against Terraform I recommend running a Terraform plan against all environments upon check in. For those unaware Terraform plan will output the list of changes Terraform will execute. When doing this in this matter each Terraform plan can be ran in a parallel against individual environments. This means we have one job running a Terraform plan against dev, one in uat, and one in production all simultaneously. The alternative is to run each sequentially which is only adds to our pipeline duration. And on that note....

Naming Stages and Jobs

Stages and Jobs must have unique names. This is an important factor when scaling out your pipelines. As such I always recommend that the name of the stage or job has information that differentiates it from each other and is something that is calculated. Environment name is a good one to use. The environment name not only is a property of the job itself; however, can be used as part of the stage and job name. This creates a uniqueness for the job. To further emphasize this, I also leverage Azure region, if applicable.

Here is an example of one such job name:

```
jobs:
- deployment: '${{ parameters.serviceName }}_app_${{ parameters.environmentName }}_${{ parameters.regionAbrv }}'
  environment: ${{ parameters.environmentName }}
```

This is taken from an webapp deployment template. The fun thing is that these parameters can be reused in other jobs within the same stage or can even be used to create the dependent jobs name.

Variable Scoping

Coping variables is a big thing to consider as well when we talk through templating pipelines. A variable can be scoped at the root, stage, or job level. It is important to understand the variables value needs to either be set to runtime or be known prior to pipeline compilation. Again think of the pipeline expansion process.

It is considered a good practice to scope the variables at the lowest level needed. This really is key when we are talking about reusing a deployment stage/job across multiple environments. Typically, this setup would be the same YAML is being ran with different variables being passed in. These variables can be loaded at the appropriate scope.

Here is a piece of a Terraform apply job which is reusable and is dynamically loading in the variable information:

```
jobs:
- deployment: terraformApply${{ parameters.environmentName }}
  displayName: Terraform Apply ${{ parameters.environmentName }}
  variables:
  - template: ../variables/azure_terraform_${{ parameters.environmentName }}_variables.yml
```

```
  - name: commandOptions
    value: ''
  environment: ${{ parameters.environmentName }}
```

For context we can have multiple environment files for terraform configuration information. Such as azure_terraform_dev_variables.yml and azure_terraform_tst_variables.yml.

Both of these files could contain:

```
variables:
  AzureSubscriptionServiceConnectionName: Example - Dev
  TerraformDirectory: Infrastructure
  TerraformStateStorageAccountContainerName: terraform-state
  TerraformStateStorageAccountName: satfdeveus
  TerraformStateStorageAccountResourceGroupName: rg-terraformState-dev-eus
```

With the appropriate Azure environment information for each. We then could reference and pass this information on to a dependent task like:

```
 - template: ../tasks/terraform_init_task.yml
        parameters:
          serviceName: ${{ parameters.serviceName }}
          TerraformDirectory: $(Pipeline.Workspace)/${{ variables.TerraformDirectory }}
          AzureSubscriptionServiceConnectionName: ${{ variables.AzureSubscriptionServiceConnectionName }}
          TerraformStateStorageAccountResourceGroupName: ${{
variables.TerraformStateStorageAccountResourceGroupName }}
          TerraformStateStorageAccountName: ${{ variables.TerraformStateStorageAccountName }}
          TerraformStateStorageAccountContainerName: ${{ variables.TerraformStateStorageAccountContainerName
}}
```

Dependencies

By it is important to understand that jobs by default will run in parallel and stages do not. A practice I have learned to assist in scaling and reusing jobs across stage templates is to provide the default value for this. The default is an empty string to denote no dependencies, one dependency is accepted as a string, and multiple dependencies is passed in as an object of strings. Since an object with one string has the same behavior and expansion behavior as an object with multiple strings, I would recommend defaulting the dependsOn object value to an empty object.

This would look something like:

```
parameters:
- name: serviceName
  type: string
  default: ''
- name: environmentName
  type: string
- name: regionAbrv
  type: string
```

```
  default: ''
- name: dependsOn
  type: object
  default: []


jobs:
- deployment: 'docker_${{ parameters.serviceName }}_${{parameters.environmentName }}_${{
parameters.regionAbrv }}_buildandpush'
  environment: ${{ parameters.environmentName }}
  dependsOn: ${{ parameters.dependsOn }}
```

This will allow this specific job to be reused by both stages that will require the job to have a dependency and ones that do not. And the best part is we won't have to go back and make a code change for this functionality since we already provided it in.

Pipeline Decorators

For those who could be wondering how to universally enforce and run a set of steps prior to pipeline execution that please check out pipeline decorators.

When to use a decorator vs a template is going to vary based on use case; however, Microsoft documentation describes the following scenario:
Pipeline decorators let you add steps to the beginning and end of every job. The process of authoring a pipeline decorator is different than adding steps to a single definition because it applies to all pipelines in an organization.

Suppose your organization requires running a virus scanner on all build outputs that could be released. Pipeline authors don't need to remember to add that step. We create a decorator that automatically injects the step. Our pipeline decorator injects a custom task that does virus scanning at the end of every pipeline job

If you or your organization curious on pipeline decorator's I'd encourage you to check out @JinL99 post on "Azure DevOps - Leveraging Pipeline Decorators for Custom Process Automation"

Conclusion

When starting to leverage YAML Pipelines at scale you are most likely going to gravitate towards a solution that leverages a consolidated template repository. Since we will have multiple pipelines calling the same repository it's important we get guidance on how to start and structure this new repository, so it scales with our organization and their pipeline adoption.

This post has covered how pipeline templates expand, the importance of understanding scope, provided a basic file structure and naming standard for organizational purposes, how to write, nest, maintain templates, scale jobs and stages, leverage variable templates, and when to use pipeline decorators.