Azure DevOps Pipelines: Templates Tasks and Jobs

Introduction

By now if you have been following this series then you have learned about tasks, jobs, and stages in addition to variables and environments. This post will start to tie these together with templates.

YAML templates provide Azure DevOps customers with the capabilities to quickly scale and deploy artifacts while following their organizations required steps securely and quickly. Templates are the ultimate goal in any DRY (Don't Repeat Yourself) implementation of Azure DevOps Pipelines.

In an effort not to overwhelm this post will cover task and job template design. To unlock how to use stage templates I have found it helpful to pair with YAML Object parameters. This will be a follow up post in this series as it can be a lot to take in.

What Are Templates?

Azure DevOps YAML templates are YAML files that can be reused including stages, jobs, tasks, and variables. If it helps think of a simple task, perhaps the DotNetCoreCLI task. For those unaware, this task can be used to build and publish a DotNetCore project, among other things. In this example we are going to use it to publish an artifact.

Now a given code deployment may involve multiple projects. Without templates we'd have to code this task repeatedly for each project in the codebase. This provides a prime opportunity to use a template. We can define this task once and resue it by passing in various parameters for each project we would like to publish.

In addition to increasing the readability of our pipeline YAML file, templates also help improve long term maintainability of our pipelines. Notice that this specific task is currently on version 2, this is evident with the @2 on the end of the task. Now whenever @3 is released, we need to only update the template file as opposed to manually updating each instance of the task across every repository that uses this task.

To help illustrate this here is an example of leveraging the DotNetCoreCLI task both with and without templates:

Without:

```yaml
- task: DotNetCoreCLI@2
  displayName: 'Publish the project - $(buildConfiguration)'
  inputs:
    command: 'publish'
    projects: '**/*.csproj'
    publishWebProjects: false
    arguments: '--no-build --configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)/$(buildConfiguration)'
    zipAfterPublish: true
```

With:

Calling task within the job:

```yaml
- template: ../tasks/dotnetcore_cli_publish_task.yml
  parameters:
    zipAfterPublish: ${{ parameters.zipAfterPublish}}
    arguments: '--configuration ${{ parameters.buildConfiguration }} --output ${{variables.dropLocation}} ${{ parameters.publishArguments }}'
    projectPath: '${{ variables.projectPath }}/**/*.csproj'
    publishWebProject: ${{ parameters.publishWebProject }}
```

Dotnetcore_cli_publish_task.yml

```yaml
parameters:
    projectPath: ''
    arguments: ''
    publishWebProject: true
    zipAfterPublish: true

steps:
- task: DotNetCoreCLI@2
  displayName: 'dotnet publish'
  inputs:
    command: publish
    publishWebProjects: ${{ parameters.publishWebProject }}
    projects: ${{parameters.projectPath}}
    arguments: ${{parameters.arguments}}
    zipAfterPublish:  ${{ parameters.zipAfterPublish }}

```

The `../tasks/dotnetcore_cli_publish_task.yml` points to the location of the YAML template file.

Design for Templating Up

“Templating Up” is the methodology I use when designing and creating templates. The thought is to create a template on each basic step or task. Once all the task templates have been created, then create a job template calling all those tasks. Lastly create stage templates that will call the desired job templates.

Tasks

To illustrate this let’s build on the previous step where we have a task for publishing a dotnet project. What if we also require a specific version of the dotnet SDK to be installed on the agent?

Similarly, we'd create a new file for the dotnet sdk task template. Perhaps it would look something like:

dotnet_sdk_task.yml

```yaml
parameters:
- name: sdkVersion
  type: string
  default: '3.1.x'

steps:
- task: UseDotNet@2
  displayName: 'Use .NET SDK v${{ parameters.sdkVersion }}'
  inputs:
    packageType: 'sdk'
    version: ${{ parameters.sdkVersion }}
    includePreviewVersions: true

```

Let's also do the same for a dotnet build task since publishing projects that don't compile is something we usually try to avoid.

dotnetcore_cli_task.yml

```yaml
parameters:
    command: ''
    projectPath: ''
    arguments: ''
steps:
- task: DotNetCoreCLI@2
  displayName: 'dotnet ${{ parameters.command}}'
  inputs:
    command: ${{ parameters.command}}
    projects: ${{ parameters.projectPath }}
    arguments: ${{ parameters.arguments}}

```

The final step of a build process should be the publication of the artifacts to the ADO pipeline. This will ensure our tasks will produce a deployable artifact for future pipeline stages and jobs to consume.
So now, if keeping track, we have the following steps to publish our project:

1. Install required SDK
2. Build Project via DotNetCoreCLI
3. Publish Project via DotNetCoreCLI
4. Publish Pipeline Artifact

Jobs

What we have here is a series of steps that can apply to any number of DotNetCore projects we have in our organization. As such the next step in templating up is to create a job template using these task templates.

If you read my previous post on jobs, then we realize that the job is dispatched to an agent so if building and publishing multiple projects we can build/publish projects simultaneously on different agents all using the same template:

dotnetcore_build_publish_job.yml

```yaml
parameters:
  buildConfiguration: 'Release'
  projectName: ''
  zipAfterPublish: true
  publishWebProject: true
  publishArguments: ''
  sdkVersion: ''



jobs:
- job: build_publish_${{parameters.projectName}}
  variables:
    projectName: ${{replace(parameters.projectName,'_','.')}}
    srcFilePath: 'src'
    projectPath: '$(Build.SourcesDirectory)/${{ variables.srcFilePath }}/${{ variables.projectName }}'
  steps:
  - template: ../tasks/dotnet_sdk_task.yml
    parameters:
      sdkVersion: ${{ parameters.sdkVersion }}

  - template: ../tasks/dotnetcore_cli_task.yml
    parameters:
      command: 'build'
      projectPath: '${{ variables.projectPath }}/**/*.csproj'
      arguments: '--configuration ${{ parameters.buildConfiguration }}'

  - template: ../tasks/dotnetcore_cli_publish_task.yml
    parameters:
      zipAfterPublish: ${{ parameters.zipAfterPublish}}
      arguments: '--configuration ${{ parameters.buildConfiguration }} --output ${{variables.dropLocation}} ${{ parameters.publishArguments }}'
      projectPath: '${{ variables.projectPath }}/**/*.csproj'
      publishWebProject: ${{ parameters.publishWebProject }}

  - template: ../tasks/ado_publish_pipeline_task.yml
    parameters:
      artifactName: ${{ parameters.projectName }}
      targetPath: ${{variables.dropLocation}}


```

This agent separation is key to understanding as it provides the opportunity to load a variable template scoped to the job. In this example we are not using them; however, the next post on YAML Objects with stages will heavily use these.

If you haven't noticed the solution specific arguments for these tasks have been abstracted as parameters, which provide flexibility. Things like the sdkversion and project name have been pulled out and made parameters, with defaulted values. This follows an 80/20 rule. The value for the parameter that is used 80% of the time should be the default.

This allows future developers not to require all the inputs. If a parameter is required, I tend to default to ''. This will allow templates to expand and compile and if missing could create an error at task execution or produce an empty string. This is opposed to templates throwing errors on required parameters at pipeline compilation time. This then leads to trouble shooting through layers of templates.

Storing and Naming Templates

When dealing with YAML Templates an important question probably arises around where to store them and how to maintain them. This question isn't that hard if we ask how we should store something that multiple repositories can use, has version control, and ability to be easily maintained. Microsoft recommends the answer to this being its own git repository. Pipelines can reference a repository like:

```
repositories:
- repository: templates
  type: git
  name: TheYAMLPipelineOne.git
```

This reads as:

- **repository**: name of the repository local to the job
- **type**: what kind of repository is it
- **name**: what the name of the repository is, if in a different project provide project name in addition

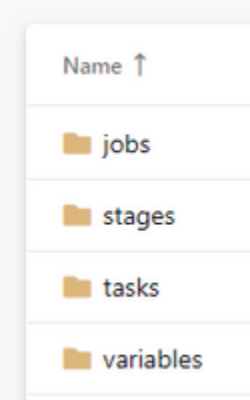
So then if we want to leverage a template in the repository it would look like:

```
- template: jobs/dotnetcore_build_publish_job.yml@templates
```

The @ sign is only required if referencing a file in a remote repository. If nesting templates in the same repository, a job that reference task template, do not use this as it will inadvertently

build a name dependency since the template will attempt to resolve the repository at the named reference.

So how to name them? Personally, I always adhere to a folder structure similar to:



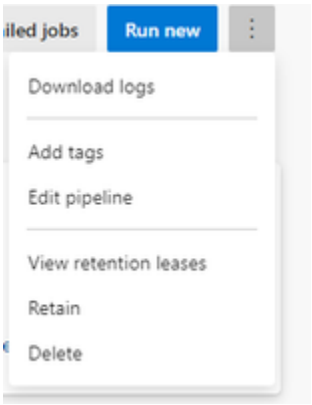
This ensures clear separation of what each template is scoped to. As for individual file names I usually go with something like technology_verb_templateType. So, in above dotnetcore(technology)_build_publish(verbs as to what it will do)_job(templateType). This again allows for quick identification of what the template is for and how to use it.

Maintaining

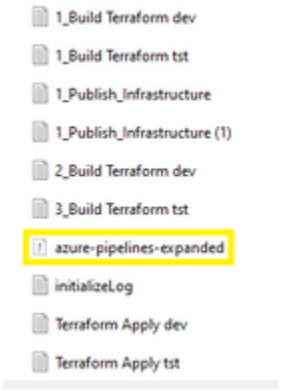
If we store these templates in a separate repository, then we need the ability to maintain changes in our deployment/build processes and tasks. This would best be accomplished via a branching strategy and testing with specifying which template repository branch to use.
If interested check out my post on Selecting Source Branches for Pipelines which was featured in Top Stories from the Microsoft DevOps Community.

Debugging

When working with templates a common question is “How can I see my Pipeline YAML?”. It’s important to understand that the templates will expand at compilation time and ADO will create one YAML pipeline file with all the tasks, jobs, stages, and variables fully expanded. This can be downloaded by selecting the pipeline and “Download logs”



This will download all your pipeline logs as a zip file. Once downloaded select “azure-pipelines-expanded.yml” file.



This will load up the pipeline with all templates and variables expanded. For additional help in troubleshooting working with templates can check out my personal blog on What to do when Azure DevOps YAML Pipelines Fail.

Conclusion

At this point we have covered how to start designing and implementing Azure DevOps templates for tasks and jobs. This should be enough to get started; however, there is still a lot to master when keeping your pipelines