Azure DevOps Pipelines: Environments and Variables

Introduction

This is part of a series on Azure DevOps pipelines, previously we discussed Azure DevOps Pipelines: tasks, jobs, and stages.

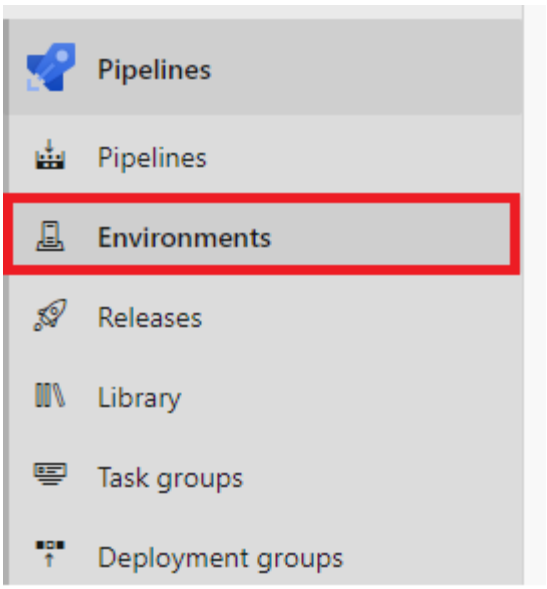
At this point you are familiar with core components of Azure DevOps pipelines (tasks, jobs, and stages) and are looking how to best leverage some additional components such as environments and variables.

Environments and variables are **two key components** when it comes to Azure DevOps pipelines **security and governance**. Additionally, if done right they assist with reusability of your pipelines.

Environments

Azure DevOps environments are defined as “a collection of resources that you can target with deployments from a pipeline”
They are much more than that though. Environments provide the ability to monitor deployments that are occurring, provide an additional level of security, and assist in tracking the state of work items in terms of where they have been deployed to.
Environments, like a lot of things withing Azure, is a very generic term.

These environments are accessed under the pipelines blade:

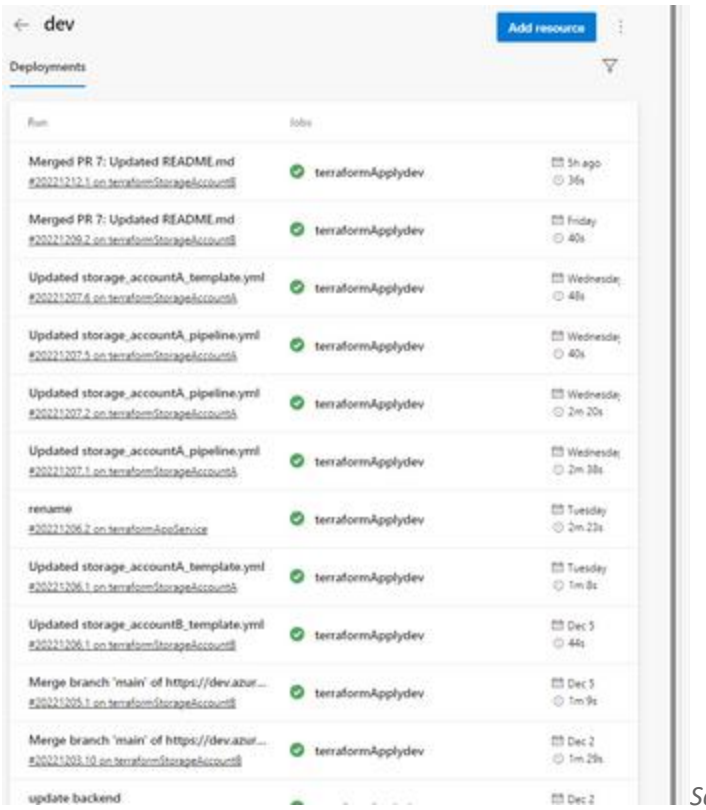


Environments under the Pipeline blade in ADO

These environments do not necessarily equivocate to a traditional dev, tst, prd model. Don't get me wrong, they certainly could, and I usually suggest doing so as a start. When and how to create an environment goes back to how the environment will be deployed and managed.

Before diving into this with more detail, it's important to understand the environment is associated with a deployment job, which is a subtype of an Azure DevOps job. Deployment jobs, in themselves, could be an additional topic; however, for this purpose the deployment job is the mechanism that registers the activity with the Azure DevOps target environment.
An environment gives users of Azure DevOps a clear prospective on what jobs have been deployed to a specific environment.

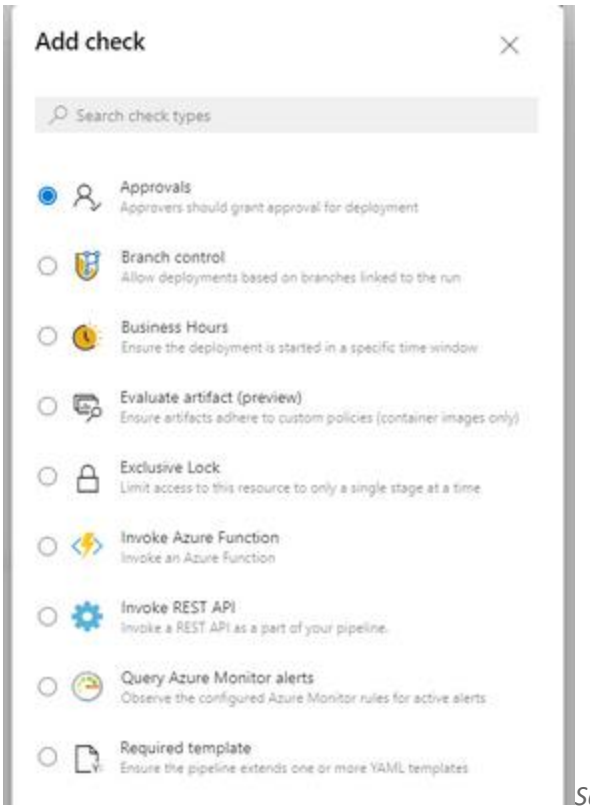
Observe below the list of jobs that have been deployed to the "dev" environment.



Screenshot of all deployment jobs ran under an environment

One can see here all the jobs and respective pipelines that have been deployed to a specified environment. This perspective gives an admin the ability to see what has been deployed in Azure DevOps to a specified scope. Which, as any administrator can attest, is a key in understanding what may have changed in an environment over a given point of time. Additionally, one can drill into the pipeline associated with the job and easily see the commits, work items, and any additional details that were included as part of the deployment.
The next key piece to leveraging environments is the ability to set gates for the environments.

A gate is a pre-approval or check before the job is dispatched to an agent(s). There are multiple options as to what could be considered a gate or approval. Traditionally manual approval is one most organization look for when going to a production environment. Alternatively, and additionally a gate could consist of invoking an Azure Function, confined to a time window, invoke a rest API, etc..



Screenshot of the various approvals and gates available to an environment

This ability provides the opportunity to enforce quality control and security around any deployments being sent out to an environment.
So, a couple guidelines when creating environments:

- All deployment jobs in the same stage should go to the same environment. This helps when rerunning and organizing approvals/gates
- Think of environments based on the audience who would be controlling gates and/or viewing jobs. Maybe a combination of app + environment makes the most sense.
- If redeploying a stage w/ environment or approval the gate or approval will be re evaluated

Variables

Azure DevOps Variables are the key when trying to optimize reuse across stages and jobs. First and foremost, variables are attached to a scope. This is the key concept when trying to best leverage them across a multi-stage and multi-environment pipeline.
A variable can be scoped at the pipeline, stage, and job level. When thinking about this a job is the sequence of tasks being dispatched to an agent, as such any variables being scoped to a job will not be inherently available to subsequent jobs since they could be running on completely different agents. Again, this is true for any variables scoped at the stage level. The same variable defined in multiple places could have different values depending on how the variable has been scoped.

Variables can be read in from:

- User Defined
- A template file
- A variable group
- System Defined

It is my experience that when creating user defined variables that there should be thought in how they will be used. Realistically user defined variables should only be leveraged if there is no potential to be reused outside of the current scope. Variables should be architected in such a fashion that they can be reused across multiple jobs, stages, and pipelines. Explicitly defining them to a set value tends to limit this and introduce more manual maintenance.

The first place I defined variables is in a template file. There will be a follow up post more focused on templates; however, not mentioning them, as an option for variables is leaving out huge functionality that is easy to add. Think of a template file as comprable to an aasettings.json, azuredeploy.parameters.json, or terraform.var file. This file SHOULD NOT CONTAIN SECRETS OR SENSITVE INFORMATION. Remeber this YAML will be stored in source control which is different than classic releases.
Defining a variable template file is easy.

All that is required is a `variable` block and the rest is key value mapping:

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

Example is taken from TheYAMLPipelineOne repo.

Referencing these values can be done so via:

```
${{variables.TerraformDirectory}}
```

The important thing when using a YAML variable template is the name. I highly recommend the name of the variable file matches the target environment as described above. This means for a dev environment perhaps the YAML variable file name will be `dev_variables.yml`.
Using this pattern, the variable file should be called within a job responsible for deploying to dev.

Here is an example of such a practice:

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

This will mean we can call the appropriate variable template corresponding to the target environment reusing the same parameter.
If needing to store a secret, then I would suggest leveraging Azure DevOps Variable Groups. Variable groups will have the added security having to explicitly grant pipelines access to them, the inability to retrieve secrets after they are stored, and Azure DevOps itself has another layer of security as to whom can edit or add variables.

Referencing a variable group in YAML is also straightforward by leveraging the 'group' keyword. I would also recommend the practice of appending or prefixing the variable group with environment name. This will give you the flexibility to load the associated variable group and variables at the requested scope:

```
variables:
- group: my-variable-group_${{ parameters.environmentName }}
```

Once loaded then the individual properties can be accessed as `$(variableName)`. Notice the difference in syntax that the variable is $() format. This means the variable will be accessed at macro level right before runtime. This is a preferred way if storing secrets as variables. More on that below.

Sometimes the need may arise to evaluate one of the built in Azure DevOps variables. A common request I come across is determining if the pipeline is executing against the trunk or release branch of the repository. This can be accommodate leveraging the 'Build.SourceBranch' variable.

This would be accomplished with:

```
- ${{ if eq(variables['Build.SourceBranch'], 'refs/heads/main')}}:
```

Here we will do the next steps if the value in the `Build.SourceBranch` variable is the main branch.

Variable Syntax

This is one of the most common items we run into when helping customers is how and when to define variables as `$(variableName)`, `${{variables.variableName}}`, or `$[variables.variableName]`.

Personally, I recommend defaulting to ${{variables.variableName}}. This will make the variable insert at compilation time. Compilation time is when the pipeline execution instance is created. I find this easy to debug if a variable is incorrectly passed since the variable will be populated in the log files. Another added benefit is when authoring pipelines can quickly tell if the referenced value name is a variable and not a parameter.

Now $(variableName) is the next type I typically use. This is used for secrets as discussed above. Additionally, since this variable will expand at task execution runtime, it will be used for all built in variables. Since we typically won't know these variable values until the pipeline executes.

For summary here is a chart from Microsoft docs:

Syntax	Example	When is it processed?	Where does it expand in a pipeline definition?	How does it render when not found?
macro	`$(var)`	runtime before a task executes	value (right side)	prints `$(var)`
template expression	`${{ variables.var }}`	compile time	key or value (left or right side)	empty string
runtime expression	`$[variables.var]`	runtime	value (right side)	empty string

Table illustrating the various ways and required syntax for variable definition

Conclusion

The combination of Azure DevOps target environments and variables can be a powerful tool to yield when expanding and creating Azure DevOps Pipelines. Having their names align can lead to properly scoping variables to jobs associated with the corresponding environments. Furthermore, environments add an additional level of control via gates and approvals. While variables add the extra functional of being user defined, defined in a template file, variable group, and expose built in variables from Azure DevOps itself. In a future post we will dive deeper into how to stitch together templates together to really accelerate your Azure Pipeline creation and maintenance.