**Agenda: Manage Pipeline Infrastructure**

- About Agent Pools
- About Agents
    - Microsoft Hosted Agents
    - Self Hosted Agents
- Pipeline Demands
- Jobs Deep Dive
    - Deployment Jobs
        - Runonce Deployment Strategy
        - Rolling Deployment Strategy
        - Canary Deployment
    - Agent Pool Jobs
    - Server Jobs
    - Container Jobs
- Build and Release Retention Policies
- About Libraries
    - Variable Groups
    - Secure Files
- Manage Pipeline Resources Permissions
- Pipeline Conditions
- Service Hooks
- Using the REST API

## About Agent pools

As you know, a build agent is a piece of installable software that runs one build or deployment job at a time. To build your code or deploy your software you need at least one agent. As you add more code and people, you'll eventually need more than one agent.

**About Agent Pools:**

- Instead of managing each agent individually, you can organize agents into agent pools.
- In Azure Pipelines, agent pools are scoped to the Azure DevOps organization so you can share an agent pool across projects.
- A project agent pool provides access to an organization agent pool.
- When you create a build or release pipeline, you specify which pool it uses.
- Pools are scoped to your project so you can only use them across build and release pipelines within a project.

- To share an agent pool with multiple projects, in each of those projects, you create a project agent pool **pointing** to an organization agent pool. While multiple pools across projects can use the same organization agent pool, multiple pools within a project cannot use the same organization agent pool. Also, each project agent pool can use only one organization agent pool.

**Here are some typical situations when you might want to create self-hosted agent pools:**

1. You're a member of a project and you want to use a set of machines owned by your team for running build and deployment jobs only for your projects.
    a. Create Pool at Project Level.
2. You're a member of the infrastructure team and would like to set up a pool of agents for use in all projects.
    a. Create Pool at Organization Level
    b. Select the option to **Auto-provision corresponding agent pools in all projects**
3. You want to share a set of agent machines with multiple projects, but not all of them.
    a. Create Pool at Organization Level.
    b. **Uncheck** the option to **Auto-provision corresponding agent pools in all projects**
    c. Go to each of the other projects, and create a pool in each of them while selecting the option to Use an **existing agent pool** from the organization**.**

**Create an Agent Pool**

Organization Settings → Pipelines → Agent Pools

**Or**

Project Settings → Pipelines → Agent Pools

**Security of agent pools in organization settings:**

1. **Reader:** Members of this role can view the agent pool as well as agents. You typically use this to add operators that are responsible for **monitoring** the agents and their health.
2. **Service Account:** Members of this role can use the organization agent pool to **create a project agent pool in a project**.
3. **Administrator**
    a. In addition to all the above permissions, members of this role can **register or unregister agents from the organization agent pool**.
    b. They can also refer to the organization agent pool when creating a project agent pool in a project.
    c. Finally, they can also **manage membership** for all roles of the organization agent pool. The user that created the organization agent pool is automatically added to the Administrator role for that pool.

**Roles for AgentPools in Project Security Settings**

1. **Reader**: Members of this role can view the project agent pool. You typically use this to add operators that are responsible for **monitoring** the build and deployment jobs in that project agent pool.
2. **User**: Members of this role can use the project agent pool when authoring pipelines.
3. **Administrator**: In addition to all the above operations, members of this role can **manage membership** for all roles of the project agent pool. The user that created the pool is automatically added to the Administrator role for that pool.

## About Build Agents – Microsoft Hosted Agents

### MICROSOFT-HOSTED AGENTS

- If your pipelines are in Azure Pipelines, then you've got a convenient option to run your **jobs** using a **Microsoft-hosted agent**. With Microsoft-hosted agents, maintenance and upgrades are taken care of for you. Each time you run a pipeline, you get a fresh virtual machine. The virtual machine is discarded after one use. Microsoft-hosted agents can run jobs directly on the VM or in a container.
- For many teams this is the simplest way to run your jobs. You can try it first and see if it works for your build or deployment. If not, you can use a self-hosted agent.

**YAML VM Image Label:**

1. **windows-latest** OR windows-2019
2. vs2017-win2016
3. **ubuntu-latest** OR ubuntu-20.04
4. ubuntu-18.04
5. ubuntu-16.04
6. **macOS-latest** OR macOS-10.15
7. macOS-10.14

Pipelines will default to the Microsoft-hosted agent pool. You simply need to specify which virtual machine image you want to use.

```
jobs:
- job: Linux
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - script: echo hello from Linux
- job: Windows
  pool:
    vmImage: 'windows-latest'
  steps:
  - script: echo hello from Windows
```

**Microsoft-hosted agents offer:**

- You can add software during your build or release using **tool installer tasks**. Eg: Use .NET Core installs the specified version of .NET SDK on the agent machine.
- Provide at least **10 GB of storage** for your source and build outputs.
- Provide a free tier:
  - **Public project**: **10 free Microsoft-hosted parallel jobs** that can run for up to 360 minutes (6 hours) each time, with no overall time limit per month. Contact us to get your free tier limits increased.
  - **Private project**: **One free parallel job** that can run for up to 60 minutes each time, until you've used 1,800 minutes (30 hours) per month. You can pay for additional capacity per parallel job. Paid parallel jobs remove the monthly time limit and allow you to run each job for up to 360 minutes (6 hours).
- Run on Microsoft Azure general purpose virtual machines **Standard_DS2_v2**
- Run as an **administrator** on Windows and a **passwordless sudo** user on Linux.

**Microsoft-hosted agents do not offer:**

- The ability to sign in (Remote Desktop Login)
- The ability to drop artifacts to a UNC file share (//servername/drive/folder).
- The ability to run XAML builds. (Xamarin / WPF / UWP Applications)
- Potential performance advantages that you might get by using self-hosted agents which might start and run builds faster.

| **Image** | **Classic Editor Agent Specification** | **YAML VM Image Label** | **Included Software** |
|---|---|---|---|
| Windows Server 2019 with Visual Studio 2019 | *windows-2019* | windows-latest OR windows-2019 | Link |
| Windows Server 2016 with Visual Studio 2017 | *vs2017-win2016* | vs2017-win2016 | Link |
| Ubuntu 20.04 | *ubuntu-20.04* | ubuntu-latest OR ubuntu-20.04 | Link |
| Ubuntu 18.04 | *ubuntu-18.04* | ubuntu-18.04 | Link |
| Ubuntu 16.04 | *ubuntu-16.04* | ubuntu-16.04 | Link |
| macOS X 10.15 | *macOS-10.15* | macOS-latest OR macOS-10.15 | Link |
| macOS X 10.14 | *macOS-10.14* | macOS-10.14 | Link |

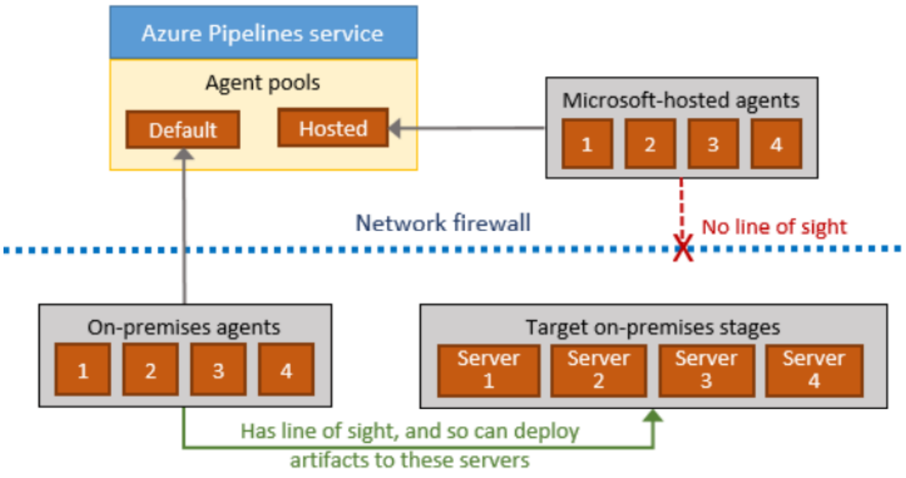
## Self Hosted Agents

- An agent that you set up and manage on your own to run jobs is a **self-hosted agent**.

- Self-hosted agents give you more control to install dependent software needed for your builds and deployments.
- Also, machine-level caches and configuration persist from run to run, which can boost speed.
- You can install the agent on Linux, macOS, or Windows machines. You can also install an agent on a Docker container.

Agents run on a separate server and install and execute actions on a deployment target. By having an agent installed in this way, you need to have permissions to access the target environment. The target environment needs to accept incoming traffic. When you use Hosted agents, running in the cloud, you cannot use these agents to deploy software on your on-premises server.

To overcome this issue, you can run agents in your local network that communicate with the central server



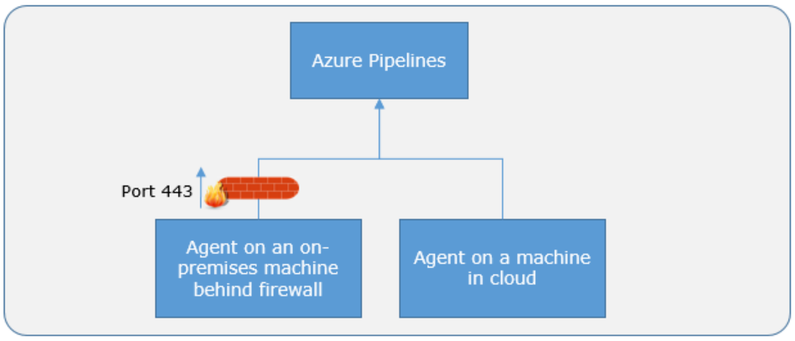
**To Create Self-hosted Windows agents**

**Step1: Make sure your machine has these prerequisites:**

- Windows 7, 8.1, or 10 (if using a client OS)
- Windows 2008 R2 SP1 or higher (if using a server OS)
- PowerShell 3.0 or higher
- .NET Framework 4.6.2 or higher

Recommended:

- Visual Studio build tools (2015 or higher)


Azure Pipelines
Port 443
Agent on an on-premises machine behind firewall
Agent on a machine in cloud

To ensure your organization works with any existing firewall or IP restrictions, ensure that **dev.azure.com** and ***.dev.azure.com** are allowed domains.

In addition, we recommend you **open port 443** to all traffic on these IP addresses and domains.

**IPv4 ranges**

- 13.107.6.0/24
- 13.107.9.0/24
- 13.107.42.0/24
- 13.107.43.0/24

**IPv6 ranges**

- 2620:1ec:4::/48
- 2620:1ec:a92::/48
- 2620:1ec:21::/48
- 2620:1ec:22::/48

About Firewall permissions – Read here: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/v2-windows?view=azure-devops#im-running-a-firewall-and-my-code-is-in-azure-repos-what-urls-does-the-agent-need-to-communicate-with

**Step2: Authenticate with a personal access token (PAT)**

1. Azure DevOps → **Profile** → Personal access tokens, and then select **+ New Token**.
2. Name your token, **select the organization** where you want to use the token, and then choose a lifespan for your token.
3. Select the **scopes** for this token to authorize for ***your specific tasks***.
4. When you're done, make sure to **copy** the token. You'll use this token as your password.

**Step3: Download and configure the agent**

1. Log on to the machine using the account for which you've prepared permissions as explained above.
2. In your web browser: **Organization Settings → Agent pools → Default → Agents Tab**
3. **Get the agent** dialog box → Windows Tab → **Download** the agent
4. Open PowerShell **in Administrator mode**

5. Unpack the agent into the directory of your choice.
6. Then run **config.cmd**.

   This will ask you a series of questions to configure the agent.

   a. When setup asks for your server URL, for Azure DevOps Services, answer https://dev.azure.com/{your-organization}.
   b. When setup asks for your authentication type, choose **PAT**. Then paste the PAT token you created into the command prompt window

**Step 4: Run the agent**

7. **Run interactively**: If you configured the agent to run interactively, to run it:

```
PS c:\agent:>.\run.cmd
```

**Run as a service**: If you configured the agent to run as a service, it starts automatically. You can view and control the agent running status from the services snap-in. Run **services.msc** and look for one of:

- "Azure Pipelines Agent (*name of your agent*)".
- "VSTS Agent (*name of your agent*)".
- "vstsagent.(*organization name*).(*name of your agent*)".

**To Remove and re-configure an agent:**

```
PS c:\agent>.\config.cmd remove
```

More About Agents: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents

## Pipeline Demands

Not all agents are the same. We've seen that they can be based on different operating systems, but they can also have different dependencies installed. To describe this, every agent has a set of capabilities configured as name-value pairs. The capabilities such as machine name and type of operating system that are automatically discovered, are referred to as **system capabilities**. The ones that you define are called **user capabilities.**
Some tasks won't run unless one or more demands are met by the agent. For example, the Visual Studio Build task demands that **msbuild** and **visualstudio** are installed on the agent.

**Manually entered demands**

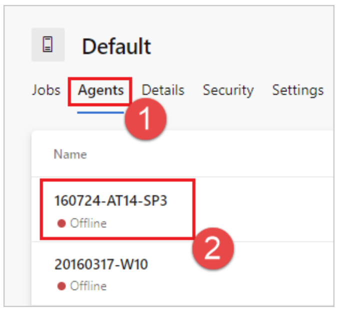
You might need to use self-hosted agents with special capabilities. For example, your pipeline may require **SpecialSoftware** on agents in the Default pool. Or, if you have multiple agents with different operating systems in the same pool, you may have a pipeline that requires a Linux agent.

**To add a single demand to your YAML build pipeline, add the demands: line to the pool section.**

```
job: Build
pool:
```

```
name: Default
demands:
 - SpecialSoftware  # Check if SpecialSoftware capability exists
 - Agent.OS -equals Linux # Check if Agent.OS == Linux
```

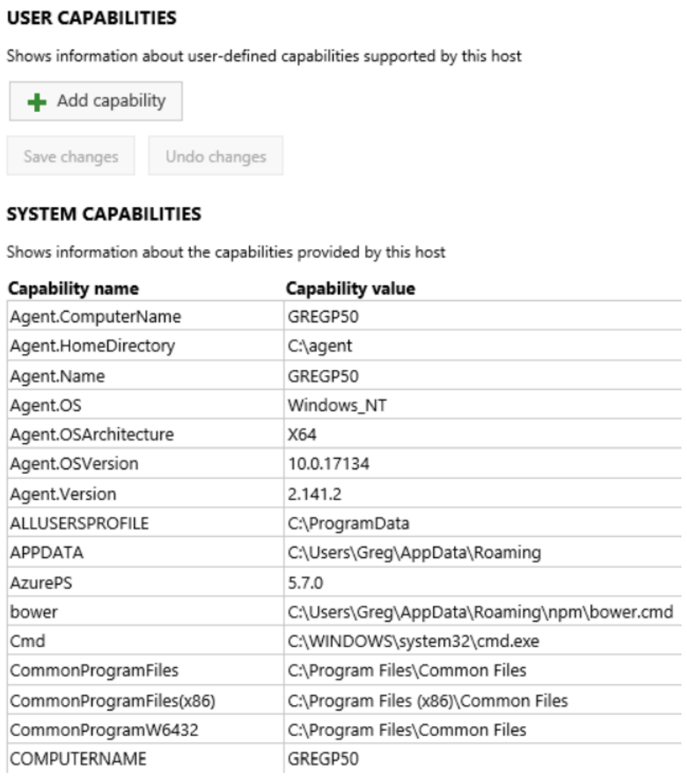
1. **Organization Settings** → Agent Pools → Default
2. Switch to Agents Tab → Select the desired agent



3. Choose the **Capabilities** tab → Add a new capability

4. Add something like the following entry:

| **First box** | **Second box** |
|---|---|
| SpecialSoftware | C:\Program Files (x86)\SpecialSoftware |
| Key1 | Value1 |


**USER CAPABILITIES**

Shows information about user-defined capabilities supported by this host

**SYSTEM CAPABILITIES**

Shows information about the capabilities provided by this host

| Capability name | Capability value |
|---|---|
| Agent.ComputerName | GREGP50 |
| Agent.HomeDirectory | C:\agent |
| Agent.Name | GREGP50 |
| Agent.OS | Windows_NT |
| Agent.OSArchitecture | X64 |
| Agent.OSVersion | 10.0.17134 |
| Agent.Version | 2.141.2 |
| ALLUSERSPROFILE | C:\ProgramData |
| APPDATA | C:\Users\Greg\AppData\Roaming |
| AzurePS | 5.7.0 |
| bower | C:\Users\Greg\AppData\Roaming\npm\bower.cmd |
| Cmd | C:\WINDOWS\system32\cmd.exe |
| CommonProgramFiles | C:\Program Files\Common Files |
| CommonProgramFiles(x86) | C:\Program Files (x86)\Common Files |
| CommonProgramW6432 | C:\Program Files\Common Files |
| COMPUTERNAME | GREGP50 |

**Note:** Please read the System capabilities.

Microsoft-hosted agents don't display system capabilities as they have pre-defined list of softwares

https://docs.microsoft.com/en-in/azure/devops/pipelines/agents/hosted

**You can specify certain demands that the agent must meet.**

Edit Build Pipeline → **Options Tab** → Under Demands → +Add

## Jobs Deep Dive

You can organize your pipeline into jobs. Every pipeline has at least one job. A job is a series of steps that run sequentially as a unit. In other words, a job is the smallest unit of work that can be scheduled to run.

Your pipeline may have multiple stages, each with multiple jobs. In that case, use the stages keyword.

```
stages:
- stage: A
  jobs:
  - job: A1
  - job: A2

- stage: B
  jobs:
  - job: B1
  - job: B2
```

**Deployment Job:** If the primary intent of your job is to deploy your app (as opposed to build or test your app), then you can use a special type of job called **deployment job**.

```
- deployment: string       # instead of job keyword, use deployment keyword
  pool:
    name: string
    demands: string | [ string ]
  environment: string
  strategy:
    runOnce:
      deploy:
        steps:
        - script: echo Hi!
```

Although you can add steps for deployment tasks in a job, we recommend that you instead use a **deployment job**.

A deployment job has a few benefits. For example, you can deploy to an environment, which includes benefits such as being able to see the history of what you've deployed.

To stop downloading artifacts, use **- download: none** or choose specific artifacts to download by specifying Download Pipeline Artifact task.

**Types of Jobs**

- **Agent pool jobs** run on an agent in an agent pool.
- **Server jobs (Agentless jobs)** run on the Azure DevOps Server.
- **Container jobs** run in a container on an agent in an agent pool.

**Agent Pool Jobs:**

These are the most common type of jobs and they run on an agent in an agent pool. Use demands to specify what capabilities an agent must have to run your job.

```
pool:
  name: myPrivateAgents
  demands:
  - agent.os -equals Windows
  - anotherCapability -equals somethingElse
steps:
- script: echo hello world
```

**Server jobs (Agentless jobs)**

Tasks in a server job are orchestrated by and executed on the server (Azure Pipelines or TFS). A server job does not require an agent or any target computers. Only a few tasks are supported in a server job at present.

Example: ManualValidation Task

```
jobs:
- job: string
  pool: server
```

**Job Dependencies:**

When you define multiple jobs in a single stage, you can specify dependencies between them.

Example jobs that build sequentially:

```
jobs:
- job: Debug
  steps:
  - script: echo hello from the Debug build
- job: Release
```

```
  dependsOn: Debug
  steps:
  - script: echo hello from the Release build
```

Example jobs that build in parallel (no dependencies):

```
jobs:
- job: Windows
  pool:
    vmImage: 'vs2017-win2016'
  steps:
  - script: echo hello from Windows
- job: macOS
  pool:
    vmImage: 'macOS-10.14'
  steps:
  - script: echo hello from macOS
- job: Linux
  pool:
    vmImage: 'ubuntu-16.04'
  steps:
  - script: echo hello from Linux
```

**Job Workspace:**

When you run an agent pool job, it creates a workspace on the agent. The **workspace is a directory** in which it downloads the source, runs steps, and produces outputs. The workspace directory can be referenced in your job using **Pipeline.Workspace** variable.

Under this, various sub-directories are created:

- **Build.SourcesDirectory** is where tasks download the application's source code.
- **Build.ArtifactStagingDirectory** is where tasks download artifacts needed for the pipeline or upload artifacts before they are published.
- **Build.BinariesDirectory** is where tasks write their outputs.
- **Common.TestResultsDirectory** is where tasks upload their test results.

**When you run a pipeline on a self-hosted agent**, by default, none of the sub-directories are cleaned* in between two consecutive runs. As a result, you can do **incremental builds and deployments**, provided that tasks are implemented to make use of that.

You can override this behavior using the workspace setting on the job.

```
- job: myJob
  workspace:
    clean: outputs | resources | all # what to clean up before the job runs
```

- outputs: Delete Build.BinariesDirectory before running a new job.
- resources: Delete Build.SourcesDirectory before running a new job.
- all: Delete the entire Pipeline.Workspace directory before running a new job.

*Note: $(Build.ArtifactStagingDirectory) and $(Common.TestResultsDirectory) are **always deleted** and recreated prior to every build regardless of any of these settings.

## Build and Release Retention Policies

Build Pipeline => **Run**

Release Pipeline => **Release**

YAML Pipeline => **Run**

Retention policies are used to configure how long **runs and releases** are to be retained by the system.

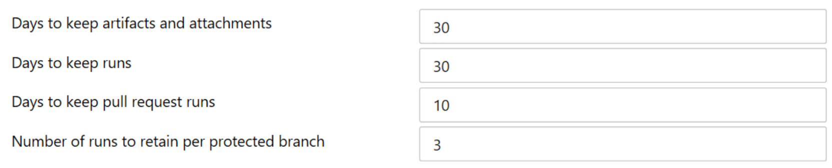
The primary reasons to delete older runs and releases are to conserve storage and reduce clutter.

The main reasons to keep runs and releases are for audit and tracking.

The following information is deleted when a run is deleted:

- Logs
- All artifacts
- All symbols
- Binaries
- Test results
- Run metadata

**Run retention: Project Settings → Pipelines → Settings**



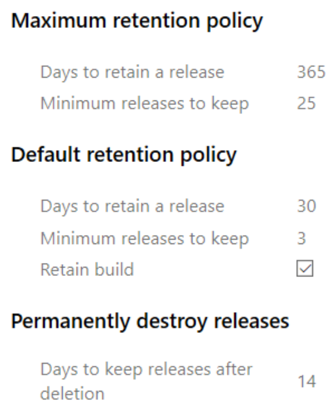
| | |
|---|---|
| Days to keep artifacts and attachments | 30 |
| Days to keep runs | 30 |
| Days to keep pull request runs | 10 |
| Number of runs to retain per protected branch | 3 |

**Ranges:**

- Artifacts, symbols and attachments range = 1 to 60
- Runs Range = 30 to 731
- Pull request runs Range = 1 to 30
- Number of recent runs to retain per pipeline = 0 to 50

**Global Release Retention: Project Settings → Pipelines → Release Retention**

- The release retention policies for a release pipeline determine how long a **release and the run linked** to it are retained.
- The retention timer on a release is reset every time a release is modified or deployed to a stage.
- The minimum number of releases to retain setting takes precedence over the number of days.


**Maximum retention policy**

| | |
|---|---|
| Days to retain a release | 365 |
| Minimum releases to keep | 25 |

**Default retention policy**

| | |
|---|---|
| Days to retain a release | 30 |
| Minimum releases to keep | 3 |
| Retain build | ✓ |

**Permanently destroy releases**

| | |
|---|---|
| Days to keep releases after deletion | 14 |

Note that you can view but not change these settings for your project.

**We can override the above settings: Select a Release Pipeline → Edit → Retension Tab → Select Stage…**

**Interaction between build and release retention**

- The build linked to a release has its own retention policy, which may be shorter than that of the release. If you want to retain the build for the same period as the release, set the **Retain associated artifacts** checkbox for the appropriate **stages**. This overrides the retention policy for the build, and ensures that the artifacts are available if you need to redeploy that release.
- When you delete a release pipeline, delete a release, or when the retention policy deletes a release automatically, the retention policy for the associated build will determine when that build is deleted.

## About Library

- *Library* is a collection of *shared* build and release assets for a project. Assets defined in a library can be used in multiple build and release pipelines of the project.
- It contains two types of assets:
  1. **variable groups**
  2. **secure files**.

**Working with Variables:**

Azure Pipelines supports three different ways to dereference variables: **macro, template expression, and runtime expression**. Each syntax can be used for a different purpose and has some limitations.

**Note:** If there's no variable by that name, then the macro expression is left unchanged. For example, if $(var) cannot be replaced, $(var) won't be replaced by anything.

**Variable Scopes**

In the YAML file, you can set a variable at various scopes:

- At the **root level**, to make it available to all jobs in the pipeline
- At the **stage level**, to make it available only to a specific stage
- At the **job level**, to make it available only to a specific job

Variables at the job level override variables at the root and stage level. Variables at the stage level override variables at the root level.

```
variables:
  global_variable: value    # this is available to all jobs

jobs:
- job: job1
  pool:
    vmImage: 'ubuntu-16.04'
  variables:
    job_variable1: value1    # this is only available in job1
  steps:
  - bash: echo $(global_variable)
  - bash: echo $(job_variable1)
  - bash: echo $JOB_VARIABLE1 # variables are available in the script environment too

- job: job2
  pool:
    vmImage: 'ubuntu-16.04'
  variables:
    job_variable2: value2    # this is only available in job2
  steps:
  - bash: echo $(global_variable)
  - bash: echo $(job_variable2)
  - bash: echo $GLOBAL_VARIABLE
```

**To Create (Classic) Release Pipeline and Stage Variables**

You define and manage these variables in the **Variables** tab in a release pipeline. In the **Release** Pipeline Variables page, open the **Scope drop-down list** and select "**Release**" or the required **stage.**

**Note:** Store sensitive values in a way that they cannot be seen or changed by users of the release pipelines.

Designate a configuration property to be a secure (secret) variable by selecting the 🔒 (padlock) icon next to the variable.

**To reference a variable in YAML**, prefix it with a dollar sign and enclose it in parentheses. For example: $(sec)

**To use a variable in a script**, use environment variable syntax.

Replace . and space with _, capitalize the letters, and then use your platform's syntax for referencing an environment variable. Examples:

Batch script: %SEC%

PowerShell script: ${env:SEC}

Bash script: $(SEC)

**Set Secret Variables (YAML)**

You should not set secret variables in your YAML file. Instead, you should set them in the pipeline editor using the web interface.

1. Navigate to **YAML file** → Top right click on Vertical … → **Variables** Button.
2. Add or update the variable.
3. Check Keep this value **secret** to store the variable in an encrypted manner.
4. Save the pipeline.

Secret variables are encrypted at rest with a 2048-bit RSA key.

Secrets are available on the agent for tasks and scripts to use **(so be careful about who has access to alter your pipeline)**.

The following example shows how to use a secret variable called mySecret from a script.

```
steps:
- powershell: |
  # Using an input-macro:
          Write-Host "This works: $(mySecret)"
```

Output:

```
This works: ***
```

**Variable Groups**

**Step 1: Create Variables Groups**

1. Pipeline → **Library** → Variable Groups Tab → Click +Variable Group
2. Set Variable group name = "**DemoGroupName**"
3. Add the following Variables
    a) AppServiceName = DssDemoApp
    b) PreProductionSlotName = "PreProd"
    c) ResourceGroupName = "DemoRG"

**Step 2.1: Use Variables in Tasks (Classic)**

4. Pipelines → Releases → **Edit** Release Pipeline → Select **Production Stage** Tasks (Tasks tab)
5. Ensure that all the tasks you intend to include do not contain any linked parameters. The easy way to do this is to choose **Unlink all** in the settings panel for the entire process.
6. Select Task **Deploy Azure App Service** → Set App Service name = $(AppServiceName), $Resource Group= $(ResourceGroup), Slot=$(PreProductionSlotName)
7. Select Task **Swap Slots**: Set App Service name = $(AppServiceName), $Resource Group= $(ResourceGroup), Source Slot=$(PreProductionSlotName)

**Variable Groups in YAML**

This example shows how to reference a **variable group** in your YAML file and also add variables within the YAML.

There are two variables used from the variable group: **user** and **token**. The **token variable is secre**t and is mapped to the environment variable $env:MY_MAPPED_TOKEN so that it can be referenced in the YAML.

```
variables:
  devopsAccount: contoso
```

```
variables:
- group: 'my-var-group' # variable group
- name: 'devopsAccount' # new variable defined in YAML
  value: 'contoso'
- name: 'projectName' # new variable defined in YAML
  value: 'contosoads'

steps:
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: |
      Write-Host $(user)
      Write-Host $env:USER
      Write-Host $(env:MY_MAPPED_TOKEN)
  env:
    MY_MAPPED_TOKEN: $(token) # Maps the secret variable $(token) from my-var-group
```

**Note:** If you use both variables and variable groups, you'll have to use name/value syntax for the individual (non-grouped) variables:

**List of Predefined variables**

https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables

**Secure Files**

- Use the **Secure Files** library to store files such as signing certificates, Apple Provisioning Profiles, Android Keystore files, and SSH keys on the server without having to commit them to your source repository.
- The contents of the secure files are encrypted and can only be used during the build or release pipeline by referencing them from a task.
- The secure files are available across multiple build and release pipelines in the project based on the security settings.
- There's a size limit of 10 MB for each secure file.

Use the **Download Secure File** task to consume secure files within a Build or Release Pipeline.

This example downloads a secure certificate file and installs it to a trusted certificate authority (CA) directory on Linux:

```
- task: DownloadSecureFile@1
  name: caCertificate
  displayName: 'Download CA certificate'
  inputs:
    secureFile: 'Demo.txt'

- script: |
    echo Installing $(caCertificate.secureFilePath) to the trusted CA directory...
    sudo chown root:root $(caCertificate.secureFilePath)
    sudo chmod a+r $(caCertificate.secureFilePath)
    sudo ln -s -t /etc/ssl/certs/ $(caCertificate.secureFilePath)
```

## Manage Pipeline Resources Permissions

**What is a resource?**

A resource is anything used by a pipeline that lives outside the pipeline itself.

Examples include:

- Secure files
- Variable groups
- Service connections
- Agent pools
- Other repositories

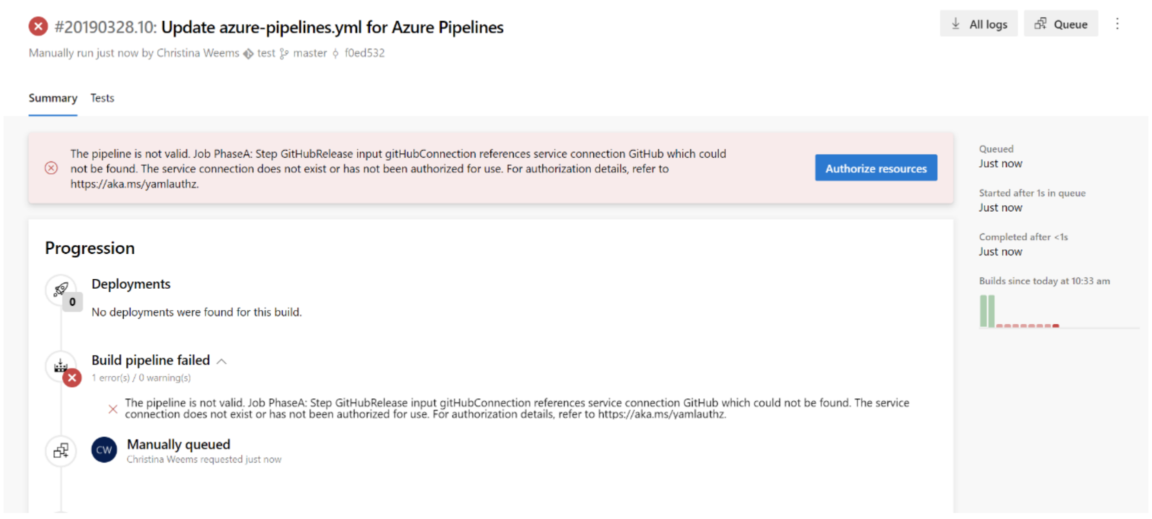
- Containers

Resources must be authorized before they can be used. A resource owner controls the **users and pipelines** that can access that resource.

**If you create a pipeline with the classic editor**, then the **user** **creating** or **editing** the pipeline must be given **User** role on that resource in order to be able to refer to that resource in the pipeline. A **resource** administrator can add users by navigating to administration page of the corresponding resource and selecting **Security** for that resource.

**If you use YAML pipelines**, then the pipeline must directly be authorized to use the resource. There are multiple ways to accomplish this.

1. Navigate to the administration experience of the resource. For example, variable groups and secure files are managed in the **Library** page under **Pipelines**. Agent pools and service connections are managed in **Project settings**. Here you can authorize **all pipelines** to be able to access that resource. This is convenient if you do not have a need to restrict access to a resource - for e.g., test resources.
2. When you create a pipeline **for the first time**, all the resources that are referenced in the YAML file are **automatically authorized** for use by the pipeline, provided that you are a member of the **User** role for that resource. So, resources that are referenced in the YAML file at pipeline creation time are automatically authorized.
3. When you **make changes** to the YAML file and add additional resources (assuming that these not authorized for use in all pipelines as explained above), then the build fails with a resource authorization error that is similar to the following: Could not find a <resource> with name <resource-name>. The <resource> does not exist or has not been authorized for use.
   In this case, on the Summary page, you will see an option to authorize the resources on the failed build. If you are a member of the **User** role for the resource, you can select this option. Once the resources are authorized, you can start a new build.


#20190328.10: Update azure-pipelines.yml for Azure Pipelines
Manually run just now by Christina Weems test master f0ed532

The pipeline is not valid. Job PhaseA: Step GitHubRelease input gitHubConnection references service connection GitHub which could not be found. The service connection does not exist or has not been authorized for use. For authorization details, refer to https://aka.ms/yamlauthz.

Progression
Deployments
No deployments were found for this build.
Build pipeline failed
1 error(s) / 0 warning(s)
The pipeline is not valid. Job PhaseA: Step GitHubRelease input gitHubConnection references service connection GitHub which could not be found. The service connection does not exist or has not been authorized for use. For authorization details, refer to https://aka.ms/yamlauthz.
Manually queued
Christina Weems requested just now

Queued
Just now
Started after 1s in queue
Just now
Completed after <1s
Just now
Builds since today at 10:33 am

## Pipeline Conditions

You can specify the conditions under which each job runs. By default, a job runs if it does not depend on any other job, or if all of the jobs that it depends on have completed and succeeded.

You can customize this behavior by forcing a job to run even if a previous job fails or by specifying a custom condition.

You can specify conditions under which a step, job, or stage will run.

- **succeeded():** It's by default, only when all previous dependencies have succeeded.
- **succeededOrFailed():** Even if a previous dependency has failed, unless the run was canceled.
- **always():** Even if a previous dependency has failed, even if the run was canceled.
- **failed():** Only when a previous dependency has failed.
- Custom conditions

```
jobs:
- job: Foo
  steps:
  - script: echo Hello!
    condition: always() # this step will always run, even if the pipeline is canceled

- job: Bar
  dependsOn: Foo
  condition: failed() # this job will only run if Foo fails
```

**Custom Condition Examples**

- **Run for the master branch, if succeeding:**

  **and**(succeeded(), **eq**(variables['Build.SourceBranch'], 'refs/heads/master'))

- **Run if the branch is not master, if succeeding**

  **and**(succeeded(), **ne**(**variables['Build.SourceBranch'], 'refs/heads/master'**))
- **Run for continuous integration (CI) builds if succeeding**

  **and**(succeeded(), **in**(variables['Build.Reason'], 'IndividualCI', 'BatchedCI'))
- **Run if the build is run by a branch policy for a pull request, if failing**

  and(failed(), eq(variables['Build.Reason'], 'PullRequest'))
- **Run if the build is scheduled, even if failing, even if canceled**

  and(always(), eq(variables['Build.Reason'], 'Schedule'))

**Use a template parameter as part of a condition**

The script in this YAML file will run because parameters.doThing is false.

```
parameters:
 doThing: false


steps:
- script: echo I did a thing
 condition: and(succeeded(), eq('${{ parameters.doThing }}', false))
```

**Use the output variable from a job in a condition in a subsequent job**

You can make a variable available to future jobs and specify it in a condition. Variables available to future jobs must be marked as multi-job output variables.

```
jobs:
- job: Foo
 steps:
  - script: |
     echo "This is job Foo."
     echo "##vso[task.setvariable variable=doThing;isOutput=true]Yes" #The variable doThing is set to true
    name: DetermineResult
- job: Bar
 dependsOn: Foo
 condition: eq(dependencies.Foo.outputs['DetermineResult.doThing'], 'Yes') #map doThing and check if Yes
 steps:
  - script: echo "Job Foo ran and doThing is true."
```

Ref: azure-pipelines-agent/outputvariable.md at master · microsoft/azure-pipelines-agent · GitHub

## Service Hooks

One of the first requests many people have when working in a system that performs asynchronous actions is to have the ability to get **notifications or alerts**.

The ability to receive Alerts and notifications is a powerful mechanism to get notified about **certain events** in your system at the moment they happen.

For example, when a build takes a while to complete, probably you do not want to stare to the screen until it has finished. But, you want to know when it does.

Almost every action in the system raises an event to which you can subscribe to. When you have made a subscription, you can then select how you want the notification to be delivered.

**Service Hooks:**

Service hooks enable you to perform tasks on other services when events happen in your Azure DevOps Services projects. For example, create a card in Trello when a work item is created or send a push notification to your team's mobile devices when a build fails.

In Azure DevOps these Service Hooks are supported Out of the Box:

- **Build and Release:** AppVeyor, Bamboo, Jenkins, MyGet, Clack
- **Collaborate**: Camfire, Flowdock, HipChat, Hubot
- **Customer Support**: UserVoice, Zendesk
- **Plan and track**: Trello
- **Integrate**: Azure Service Bus, Azure Storage, **WebHooks**, Zapier

**Note:**

If the application that you want to communicate with isn't in the list of available application hooks, you can almost always use the **Web Hooks** option as a generic way to communicate. It allows you to make an **HTTP POST** when an event occurs. So, if for example, you wanted to call an Azure Function or an Azure Logic App, you could use this option.

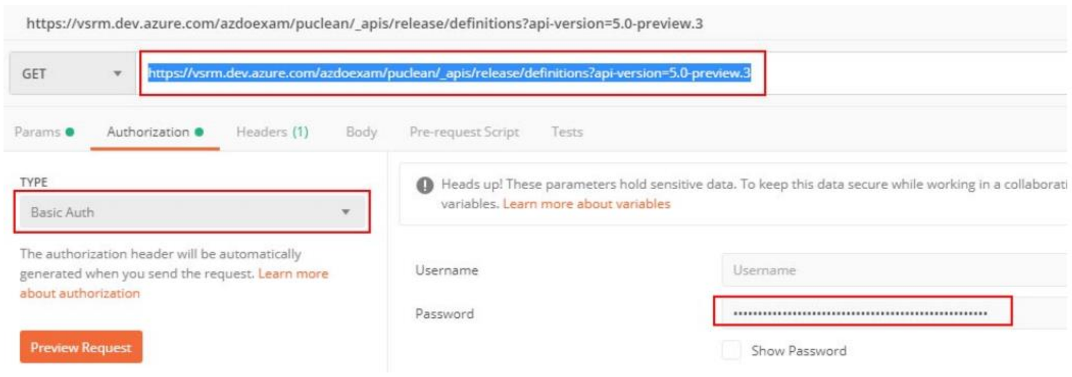
**Demo 1: Setting Up Service Hooks to Monitor the Pipeline**

Let's now take a look at how a release pipeline can communicate with other services by using service hooks.

1. Goto Project Settings → Service Hooks → **Create Subscription**
2. **Service**: From the list of available applications, click **Azure Storage → Next**
3. **Triggers:** Trigger on this type of event = **Release deployment completed**, then in the Release pipeline name select Release to all environments. For Stage, select Production. Drop down the list for Status and note the available options.
4. **Actions:** Capture the details from Azure Portal and provide the same.
5. Make sure that the test succeeded, then click Close, and on the Action page, click Finish.
6. Execute the Release Pipeline and Note that the message is posted in the queue.

## Use the REST API

1. Get a Personal Access Token
2. Use the URL https://dev.azure.com/{organization}/{project}/_apis/release/definitions?api-version=6.1-preview.1 to get all Release Definitions.
3. Download and run **Postman**
4. In the authorization tab, select **Basic Authentication** and paste the **Personal Access token** in the **password** field


https://vsrm.dev.azure.com/azdoexam/puclean/_apis/release/definitions?api-version=5.0-preview.3

GET https://vsrm.dev.azure.com/azdoexam/puclean/_apis/release/definitions?api-version=5.0-preview.3

TYPE: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborati variables. Learn more about variables

Username: Username

Password: ••••••••••••••••••••••••••••••••••••••••••••••••••••

5. Press Send and look at the results.

Note: Full REST API Documentation: https://docs.microsoft.com/en-us/rest/api/azure/devops